UNIVERSIDADE FEDERAL DA BAHIA
ESCOLA POLITÉCNICA / INSTITUTO DE MATEMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MECATRÔNICA

SANDRO SANTOS ANDRADE

SISTEMAS DISTRIBUÍDOS DE SUPERVISÃO E CONTROLE BASEADOS EM COMPONENTES DE TEMPO-REAL

Salvador
2006

SANDRO SANTOS ANDRADE

SISTEMAS DISTRIBUÍDOS DE SUPERVISÃO E CONTROLE BASEADOS EM COMPONENTES DE TEMPO-REAL

Dissertação apresentada ao Programa de Pós-Graduação em Mecatrônica, Escola Politécnica / Instituto de Matemática, Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Mecatrônica.

Orientador: Prof. Dr. Raimundo José de Araújo Macêdo.

Salvador
2006

Biblioteca Central Reitor Macêdo Costa

A553 Andrade, Sandro Santos.
Sistemas distribuídos de supervisão e controle baseados em componentes de tempo-real / Sandro Santos Andrade. - 2006.
216 f. ; il.

Inclui apêndice.
Orientador: Prof. Dr. Raimundo José de Araújo Macêdo.
Dissertação (mestrado) - Universidade Federal da Bahia, Escola Politécnica, Instituto de Matemática, 2006.

1. Programação em tempo-real. 2. Engenharia de software. 3. Framework (programa de computador). 4. Software - Reutilização. 5. Mecatrônica. I. Macêdo, Raimundo José de Araújo. II. Universidade Federal da Bahia. Escola Politécnica. III. Universidade Federal da Bahia. Instituto de Matemática. IV. Título.

CDU - 519.68
CDD - 005.13

TERMO DE APROVAÇÃO

SANDRO SANTOS ANDRADE

SISTEMAS DISTRIBUÍDOS DE SUPERVISÃO E CONTROLE BASEADOS EM COMPONENTES DE TEMPO-REAL

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Mecatrônica, Universidade Federal da Bahia, pela seguinte banca examinadora:

Raimundo José de Araújo Macêdo - Orientador

Ph.D., University of Newcastle upon Tyne, Inglaterra.
Professor Titular do Departamento de Ciência da Computação (DCC) da Universidade Federal da Bahia.

Joni da Silva Fraga

Doutor, Institut National Polytechnique de Toulouse (INPT), França.
Professor Titular do Departamento de Automação e Sistemas (DAS) da Universidade Federal de Santa Catarina.

Jean-Marie Farines

Doutor, Institut National Polytechnique de Toulouse (INPT), França.
Professor Titular do Departamento de Automação e Sistemas (DAS) da Universidade Federal de Santa Catarina.

Luciano Porto Barreto

Doutor, Université de Rennes, França.
Professor Adjunto do Departamento de Ciência da Computação (DCC) da Universidade Federal da Bahia.

Salvador, 22 de setembro de 2006

A

meus pais, Dilce e Geraldo, por terem sempre acreditado e confiado em mim.
meu filho, Benjamin, pelo lindo sorriso encorajador e pelo eterno presente da sua existência.

AGRADECIMENTOS

À vida que me foi concebida e concedida, frutificadora de tantas oportunidades para crescer e vencer desafios. Veículo para a concretização de sonhos tantos, infindáveis ... que justificam os percalços que permeiam o nosso caminho.

A minha mãe, Dilce Domingas dos Santos, pelo apoio sempre presente em tudo. A meu pai, Geraldo Galvão de Andrade Filho, pelo amor de sempre e pelas perguntas constantes de "Como vai o mestrado ?".

A Dedéa, pelo companheirismo, compreensão e amor que a fazem a luz da minha vida.

A Andréa Amorim, pelos constantes incentivos ao mundo acadêmico e pelo fruto mais precioso nesse período da minha vida.

A Raimundo José de Araújo Macêdo, pela excelente orientação e acompanhamento do trabalho, pelo zelo da relação orientando/orientador e pela compreensão e amizade de sempre.

A Marcelo Embiruçu de Souza, pela disponibilidade e pela inigualável capacidade e boa vontade em ensinar e contribuir para o sucesso do trabalho.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo apoio financeiro.

Aos professores, secretárias e colegas do Programa de Pós-Graduação em Mecatrônica (PPGM) da UFBa, pela boa convivência durante o período. Agradecimentos especiais a Socorro, Lúcia, Rebeca e Carmen, pela atenção dispensada e por tornarem tudo muito mais fácil no decorrer do projeto.

Aos professores: Herman Lepkinson, pela simpatia e pelo apoio de sempre, Leizer Schnitman, pela ajuda inicial com o CLP e com as teorias de controle, Luciano Porto Barreto, por não me deixar o TAO passar despercebido e Augusto Loureiro, pela ajuda com a tentativa de uso das placas.

A toda a equipe do *DOC (Distributed Object Computing) Group* da *University of California, Irvine* e *Washington University*, em especial a Dr. Douglas Schmidt, Gan Deng, William Otte, Nanbor Wang, Tom Ritter, Jeff Parsons, Ming Xiong e Jaiganesh Balasubramanian.

Ao colega Claudio Wakabaiashi, pelas tentativas de montagem do experimento de controle. Ao colega Luciano Coutinho, pela visita à planta da Xerox e pela troca de experiências.

Aos amigos Alírio, Ivo, Fred, Geovani, Leopoldo e Neima, pela presença, ajuda e troca de idéias.

A Fabíola Greve, João Gualberto e toda a equipe do projeto Compose, realizado pelo CPD da UFBa, pela oportunidade de realização de um excelente trabalho, responsável pela maturidade dos conceitos de engenharia de software aplicados neste trabalho.

A *Rockwell Automation* pelos *kits* didáticos, CLP's e treinamentos realizados no Centro de Capacitação em Automação Industrial da UFBa, indispensáveis para a realização deste trabalho.

A todos, meu sincero muito obrigado !

"Sem a convicção de uma harmonia íntima do Universo, não poderia haver ciência. Esta convicção é, e continuará a ser, a base de toda a criação científica. Em toda a extensão dos nossos esforços, nas lutas dramáticas entre as velhas e as novas concepções, entrevemos a ânsia eterna de compreensão, a intuição inabalável da harmonia universal, que se robustece na própria multiplicidade dos obstáculos que se oferecem ao nosso entendimento."

Albert Einstein

RESUMO

O desenvolvimento dos primeiros sistemas industriais de tempo-real foi caracterizado pela utilização de soluções proprietárias e de técnicas *ad-hoc*, com o objetivo primordial de atender os requisitos de confiabilidade e previsibilidade temporal impostos pelo ambiente. Com a evolução das tecnologias de processamento e comunicação, demandas relacionadas a distribuição, flexibilidade, extensibilidade, adaptação, uso de algoritmos inteligentes, interoperabilidade e reutilização, passaram a ser consideradas. A abordagem tradicional, caracterizada pela utilização de CLP's (Controladores Lógico-Programáveis) e de programas escritos na linguagem LADDER, constitui um entrave para a implementação dessas novas demandas, devido a restrições substanciais nas capacidades de processamento, armazenamento e conectividade. Uma alternativa é a utilização de componentes COTS (*Commercial Off-The-Shelf*) de hardware e software, reduzindo custos, facilitando questões de interoperabilidade e fazendo uso de uma infra-estrutura computacional mais poderosa. Nesse cenário, soluções extensivamente baseadas em software possibilitam a utilização de tecnologias, paradigmas e metodologias para a construção de sistemas flexíveis, reutilizáveis e interoperáveis, tais como a orientação a objetos, padrões de projeto (*design patterns*), soluções de *middleware*, *frameworks* e componentes distribuídos de software. Esta dissertação apresenta o projeto e implementação do ARCOS, uma plataforma baseada em componentes de software dedicada à construção de sistemas distribuídos de supervisão e controle, com facilidades para reutilização, interoperabilidade e especificação de restrições temporais. O ARCOS define e implementa serviços para as atividades de aquisição de dados, controle e supervisão, disponibilizando uma solução parcial e reutilizável em uma variedade de aplicações industriais. São apresentados o projeto desses serviços, com ênfase na adoção do padrão DAIS (*Data Acquisition from Industrial Systems*) para interoperabilidade, bem como as soluções relacionadas à previsibilidade temporal da plataforma. Aspectos de implementação são discutidos e são apresentadas duas aplicações de validação construídas sobre a plataforma ARCOS: um sistema supervisório para monitoramento de um reator químico e um sistema para controle PID de um piloto automático. O projeto contempla ainda a implementação de um ambiente genérico para aquisição de dados (o *DAIS Server Browser*) e de uma ferramenta de auxílio ao desenvolvimento de novas aplicações baseadas no ARCOS (o *ARCOS Assembly Tool*).

Palavras-chave: sistemas industriais de supervisão e controle, sistemas de tempo-real, desenvolvimento baseado em componentes distribuídos, *frameworks* para sistemas de tempo-real.

ABSTRACT

The development of the ancient industrial real-time systems was characterized by the use of proprietary solutions and ad-hoc techniques, which aimed to address the reliability and temporal predictability requirements imposed by the operational environment. As hardware and communication technologies became more powerful, the demands in terms of distribution, flexibility, extensibility, adaptive behaviour, intelligent algorithms, interoperability, and reusability, began to be considered when designing such industrial systems. The classical approach, characterized by the use of PLC's (Programmable Logic Controllers) and software written in the LADDER programming language, constitutes an obstacle for the implementation of such new demands, due to constraints related to computation, storage, and connectivity power. A promising alternative is the use of COTS (Commercial Off-The-Shelf) hardware and software components, which saves costs, enhances interoperability achievement, and relies on a more powerful computational infrastructure. In this scenario, sofware-intensive solutions allow for the use of technologies, paradigms, and methodologies that leverage the achievement of flexibility, reusability, and interoperability, such as the object-oriented programming, design patterns, middleware, frameworks, and distributed components. This dissertation presents the design and implementation of ARCOS, a component-based software platform devoted to the construction of supervision and control distributed systems, with facilities for reusability, interoperability, and specification of temporal constraints. ARCOS defines and implements services for activities related to industrial data acquisition, control, and supervision, providing a parcial and reusable solution for a range of industrial applications. We present the design of these services, focusing on the adoption of DAIS (Data Acquisition from Industrial Systems) standard, as well as the solutions related to the temporal predictability of our platform. We also discuss implementation issues and present two applications built atop ARCOS: a supervisory system for a chemical reactor and a system for a cruise PID control system. Our project encompass the implementation of a generic environment for industrial data acquisition (the DAIS Server Browser) and a development tool devoted to the implementation of new ARCOS-based real-time applications (the ARCOS Assembly Tool).

Keywords: supervision and control industrial systems, real-time systems, component-based software development, frameworks for real-time systems.

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviaturas e Siglas

ABEL	Allen Bradley Ethernet Library
ACE	ADAPTIVE Communication Environment
ACCORD	AspeCtual COmponent based Real-time system Development
ADAPTIVE	A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment
AMT	ARCOS Management Tool
API	Application Program Interface
ARCOS	ARquitetura para COntrole e Supervisão
AST	ARCOS Assembly Tool
CCM	CORBA Component Model
CIAO	Component-Integrated ACE ORB
CLP	Controlador Lógico-Programável
CNC	Computer Numerical Control
CORBA	Common Object Request Broker Architecture
COS	Common Object Services
COTS	Commercial Off-The-Shelf
CVS	Concurrent Versioning System
DAIS	Data Acquisition from Industrial Systems
DAnCE	Deployment And Configuration Engine
DCOM	Distributed COMponents
EDF	Earliest Deadline First
EJB	Enterprise Java Beans
FIFO	First-In First-Out
FMC	Flexible Manufacturing Cell
GIOP	General Inter-ORB Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
IPC	Inter-Process Communication
MMS	Manufacturing Message Specification
MUF	Maximum Urgency First
ORB	Object Request Broker
OSA+	Open System Architecture PLatform for Universal Services
PID	Proporcional Integral Derivativo
QoS	Quality of Service
QuO	Quality Objects
RCCF	Real-time Component Customization Framework
ReDaC	ReDedeployment and reConfiguration
RIDL	Real-time Interface Definition Language
RIOP	Real-time Inter-ORB Protocol
RMI	Remote Method Invocation
RMS	Rate Monotonic Scheduling
RPC	Remote Procedure Call
SCADA	Supervisory Control And Data Acquisition
TAO	The ACE ORB
TMO	Time-Triggered Message-Triggered Object
TMOSM	Time-Triggered Message-Triggered Object Support Middleware

USB	Universal Serial Bus
UUID	Universal Unique IDentifier
XML	eXtensible Markup Language

Sumário

Capítulo 1

Introdução

A MECATRÔNICA GANHOU LEGITIMIDADE na academia em 1996, a partir da publicação da primeira edição do *IEEE/ASME Transactions on Mechatronics.*

> *"A Mecatrônica é a integração sinérgica da engenharia mecânica com a eletrônica e o controle computadorizado inteligente, aplicada ao projeto e manufatura de produtos e processos industriais."*
>
> Auslander, 1996 [4]

A Mecatrônica integra a Engenharia Mecânica, a Engenharia Elétrica e a Ciência da Computação, com o objetivo principal de desenvolver soluções otimizadas e inteligentes para aplicação na indústria [11]. Essas três áreas constituintes surgiram e evoluíram em instantes distintos da história.

Com a Revolução Industrial, iniciada na Inglaterra na segunda metade do século XVIII, a Engenharia Mecânica se beneficia de invenções, como a máquina a vapor, e promove um considerável aumento na produtividade e queda nos preços dos produtos manufaturados. A invenção do pára-raios, por Benjamin Franklin em 1752, caracteriza o início do uso da energia elétrica em benefício do homem. A microeletrônica, entretanto, só se consolidou quase dois séculos depois, com o surgimento das válvulas a diodo, em 1940 e dos *transistors,* em 1950. Como conseqüência desses avanços, em 1964 surgem os computadores da terceira geração, baseados em circuitos impressos e na arquitetura proposta por John von Neumann.

À medida em que a Engenharia Elétrica disponibilizava soluções que permitiam a construção de máquinas cada vez mais poderosas, a Ciência da Computação estudava, dentre outros aspectos, princípios e metodologias para o desenvolvimento de sistemas computacionais cada vez mais corretos, completos, fáceis de manter e integráveis. Nota-se, todavia, que a real contribuição da Ciência da Computação para o projeto e desenvolvimento de sistemas mecatrônicos ainda é modesta, se comparada com as das outras duas áreas constituintes [85].

Nos últimos anos, as constantes evoluções das tecnologias de hardware e de comunicação têm modificado significativamente as metodologias e técnicas utilizadas no desenvolvimento de sistemas computacionais. Com a disponibilidade de recursos computacionais mais poderosos, a complexidade desses sistemas

cresce consideravelmente à medida em que novas funcionalidades passam a ser passíveis de implementação. Além dessas novas funcionalidades, requisitos tais como distribuição, flexibilidade, extensibilidade, adaptação, uso de algoritmos inteligentes, interoperabilidade e reutilização passam a ser considerados e o uso de mecanismos para gerenciar essa crescente complexidade se torna mandatório.

No ambiente industrial não é diferente: a utilização de soluções baseadas em software tem sido cada vez mais considerada no projeto e desenvolvimento de sistemas industriais. Em conjunto com os novos requisitos acima citados, a necessidade de soluções tolerantes a falhas e temporalmente previsíveis tem demandado importantes pesquisas voltadas para a aplicação de técnicas da engenharia de software nos sistemas industriais de tempo-real.

Este capítulo aborda o contexto, motivação e objetivos desta dissertação, através da apresentação do papel do software na indústria e, em particular, nos sistemas industriais de supervisão e controle.

1.1 O papel do software na indústria

A utilização, pela indústria, de soluções baseadas em software expandiu-se em 1968 quando os Controladores Lógico-Programáveis (CLP's) passaram a substituir os painéis de *relays*, utilizados até então nos processos de automação de fábricas. Esses painéis eram formados por centenas ou até milhares de dispositivos mecânicos (*relays*) conectados uns aos outros através de fios, implicando em manutenções constantes e geralmente realizadas a altos custos. O primeiro CLP utilizado comercialmente foi o *MOdular DIgital CONtroller* (MODICON) desenvolvido pela *Bedford Associates* para uso em uma grande companhia americana de produção de automóveis [35]. O objetivo era disponibilizar uma solução com manutenção facilitada e robusta em relação às intempéries do ambiente industrial.

Em meados da década de 70 surgiram os primeiros esforços para a adoção de redes de comunicação na indústria. O primeiro sistema de comunicação industrial foi o *Modbus*, utilizado pelo já conhecido MODICON, possibilitando a troca de mensagens entre CLP's e o controle de dispositivos geograficamente distantes. Neste mesmo período, os CLP's passaram a trabalhar com voltagens variáveis, sendo então utilizados em situações onde informações analógicas eram manipuladas [35]. Por outro lado, a ausência de padronizações e a constante evolução dos equipamentos dificultavam a interoperabilidade na comunicação entre CLP's. Na década de 80 surgiram os primeiros trabalhos para padronização dos protocolos de comunicação e para o desenvolvimento de ferramentas para programação de CLP's através dos PC's (*Personal Computers*), operação até então realizada por terminais dedicados de programação. Os anos 90 foram caracterizados por melhorias nos protocolos de comunicação mais utilizados e pela possibilidade de programação de CLP's em linguagens tradicionais como, por exemplo, a linguagem C.

Apesar de todos os benefícios obtidos com a criação e adoção dos CLP's, algumas características dessa

solução constituem entraves para o desenvolvimento de sistemas industriais modernos. Devido aos restritos recursos da maioria dos CLP's, em relação à capacidade de armazenamento, poder computacional e comunicação, a sua programação é geralmente realizada de forma *ad-hoc* e com o objetivo de satisfazer requisitos funcionais e de ambiente, específicos de uma situação em particular. A ausência de uso de metodologias e de técnicas da engenharia de software em soluções baseadas em CLP's é também justificada pelo papel simples e restrito desempenhado por tais sistemas, geralmente encarregados de fechar malhas de controle ou implementar sistemas de automação através de implementações mono-programadas e executadas de forma cíclica. Com a nova demanda por sistemas industriais flexíveis, extensíveis, adaptativos, inteligentes, interoperáveis e reutilizáveis, novas pesquisas vêm sendo realizadas com o intuito de aplicar técnicas da engenharia de software, já utilizadas em sistemas convencionais, nos sistemas industriais e sistemas de tempo-real [84, 85, 105]. Tais soluções, entretanto, requerem uma infra-estrutura computacional que suporte suas demandas por processamento, armazenamento (principal e secundário) e comunicação.

Uma tendência atual é a utilização de componentes de prateleira (*Commercial Off-The-Shelf* - COTS), e em particular PC's, no projeto e implementação de sistemas industriais [35]. A capacidade de processamento e armazenamento dos computadores atuais, em conjunto com as facilidades de comunicação e conectividade, tem contribuído para o surgimento de pesquisas interessadas na utilização de PC's como infra-estrutura para soluções industriais baseadas em software. Por outro lado, a baixa confiabilidade do hardware em relação às falhas e às intempéries do ambiente, aliado à imprevisibilidade temporal causada por soluções complexas de software, representam os principais desafios para a concretização dessas pesquisas.

Portanto, a disponibilização de uma infra-estrutura computacional e de comunicação mais poderosa é fator fundamental para o desenvolvimento de sistemas industriais modernos, os quais requerem metodologias e técnicas para gerenciar a complexidade causada pelos novos requisitos. A engenharia de software é a área da Ciência da Computação responsável pelo estudo e desenvolvimento de metodologias e tecnologias que propiciam a construção de sistemas complexos mantendo, ao mesmo tempo, facilidades de manutenção, reutilização e escalabilidade[1], dentre outras. Diversas tecnologias têm sido utilizadas com o objetivo de gerenciar a crescente complexidade dos sistemas e o atendimento dos novos requisitos. Pesquisas recentes [43, 44] estudam adaptações das técnicas da engenharia de software para o uso efetivo em sistemas industriais e de tempo-real, destacando o uso de *middleware* [18, 84], modelos arquiteturais [93], *frameworks* [10, 58], padrões de projeto (*design patterns*) [87], orientação a aspectos [110] e componentes de software [30, 82].

[1] Capacidade de um sistema não ter seu desempenho degradado exponencialmente no momento em que surgem novas demandas por recursos.

1.2 Componentes de software

Há muito tempo investigam-se meios para "industrializar" o processo de desenvolvimento de software. O sucesso apresentado pelas linhas industriais de montagem de automóveis, bem como a produção padronizada de placas de circuitos eletrônicos, serviram de inspiração para tentativas de adaptação dessas estratégias para uso no desenvolvimento de sistemas. Componentes de software é a tecnologia que possibilita a construção de sistemas computacionais a partir da conexão e configuração de peças de software funcionais e reutilizáveis (componentes) [45, 34, 54, 83, 99]. Alguns benefícios surgem como conseqüência imediata dessa abordagem, a saber:

- Produtividade: o levantamento de técnicas que permitem a reutilização de soluções é uma preocupação constante da engenharia de software, desde a simples modularização de programas até o uso de componentes distribuídos. A existência de um repositório de soluções, implementadas sob a forma de componentes, alavanca de forma determinante a produtividade do processo de desenvolvimento, à medida em que o desenvolvedor se limita a questões relacionadas à conexão e configuração desses componentes. Além disso, essa conexão e configuração é geralmente realizada através de ferramentas, poupando esforços de implementação.
- Qualidade: com a complexidade introduzida pelos requisitos dos sistemas modernos, implementar todas as funcionalidades requeridas se torna uma atividade custosa, demorada e propensa a erros. A reutilização de soluções validadas, corretas e robustas é ponto fundamental para a qualidade do sistema desenvolvido e para a produtividade do processo de desenvolvimento. As demandas por flexibilidade, distribuição e tolerância a falhas são requisitos não-funcionais, independentes do negócio da aplicação, e geralmente disponibilizados como componentes já implementados, validados e com alto potencial de reutilização.
- Flexibilidade: o processo de combinação de componentes é geralmente realizado através de pontos de conexão bem definidos e localizados, os quais permitem a manutenção e substituição de partes da aplicação de forma facilitada. Componentes podem ser modificados ou substituídos desde que os critérios de conexão continuem sendo atendidos.
- Escalabilidade: uma das características da tecnologia de componentes é a possibilidade de execução e conexão de componentes em um ambiente distribuído. Desta forma, novos servidores podem ser adicionados e componentes podem ser realocados nesses servidores, de modo a garantir uma boa escalabilidade da aplicação.
- Interoperabilidade: um ponto fundamental para o sucesso da tecnologia de componentes é a utilização de padronizações. Essas padronizações permitem que componentes desenvolvidos por forne-

cedores distintos interoperem sem grandes esforços para o desenvolvedor. Interoperabilidade é uma questão que permeia diversos pontos no projeto de um sistema computacional, envolvendo sistemas operacionais, redes de comunicação, linguagens de programação e arquiteturas de hardware. O grau de heterogeneidade desses pontos é fator importante para a escolha da padronização de componentes a ser utilizada no desenvolvimento do sistema.

Maximizar a reutilização de soluções e minimizar a necessidade de desenvolver novas soluções traz benefícios em relação a custo, produtividade e qualidade. Estudos sobre reutilização evidenciam que de 40% a 60% da implementação é reutilizável entre aplicações, 60% do projeto e da implementação são reutilizáveis entre sistemas de informação e somente 15% constitui implementações específicas de um determinado sistema [83, 21].

1.3 Sistemas industriais de supervisão e controle

Um ambiente industrial é geralmente composto por uma diversidade de equipamentos eletro-mecânicos integrados a soluções baseadas em computadores: sensores e atuadores promovem a interação do sistema com o ambiente industrial, braços de robôs realizam montagens e inspeções, máquinas de comando numérico (*Computer Numerical Control* - CNC) realizam manufaturas básicas, CLP's executam programas de automação e malhas de controle, redes de comunicação interligam sub-sistemas, supervisórios monitoram as atividades etc. O termo "chão de fábrica" é comumente utilizado para designar o ambiente no qual ocorre a produção e que hospeda os dispositivos envolvidos diretamente no processo. O termo "planta" refere-se a todo o conjunto de soluções, incluindo aquelas não envolvidas diretamente no processo, tais como sistemas de monitoramento, bancos de dados etc. Dentre essas soluções, destacam-se os Sistemas Industriais de Supervisão e Controle (S&C). Conforme ilustrado na figura 1.1, sistemas de S&C são caracterizados por três níveis básicos, listados a seguir.

- Aquisição de Dados: a obtenção de dados advindos do chão-de-fábrica, bem como o envio de informações para atuação na planta, são funcionalidades requeridas por qualquer sistema industrial. Para a realização dessas operações, dois fatores devem ser considerados: a previsibilidade e a heterogeneidade dos dispositivos de aquisição/atuação e dos meios de comunicação. Devido ao papel geralmente crítico desempenhado pelos sistemas de controle, as operações de aquisição e atuação devem ser realizadas de forma temporalmente previsível. Essa previsibilidade deve ser compatível com os requisitos de tempo ditados pelo ambiente que está sendo controlado. O ambiente industrial é inerentemente heterogêneo. Tipicamente, o chão-de-fábrica de uma indústria é formado por

equipamentos adquiridos de fornecedores distintos, com o intuito de adotar as melhores soluções para cada caso específico ou como conseqüência da evolução da planta ao longo do tempo. Os fornecedores geralmente são especializados em determinados equipamentos tais como: CLP's, braços de robôs, máquinas de comando numérico ou sensores. É importante que os equipamentos adquiridos e as soluções de software estejam em conformidade com padronizações abertas para comunicação, de modo a garantir a integração da planta.

- Controle: as atividades de controle são realizadas por sistemas que tentam manter uma determinada variável em um valor pré-estabelecido (*setpoint*), mesmo na presença de perturbações do ambiente. Como exemplos, pode-se citar o controle da temperatura de uma caldeira em uma planta industrial ou o controle da velocidade de um veículo. Esse controle é geralmente realizado de forma cíclica: verifica-se o valor atual da variável através de sensores, calcula-se a atuação em função da diferença entre o valor atual e o valor desejado e realiza-se a modificação na planta através de atuadores. É esperado que esta atuação conduza a variável para o valor desejado. Quando o cálculo da atuação é realizado por um computador, tem-se o controle digital automático. Dentre as diversas técnicas de controle propostas, algumas são comumente utilizadas na indústria, como o controlador PID (Proporcional Integral Derivativo); outras constituem pesquisas recentes da área, tais como os controladores multi-variáveis, adaptativos e baseados em algoritmos inteligentes. De um modo geral, os sistemas de S&C disponíveis atualmente apresentam baixa flexibilidade em relação a qual estratégia de controle pode ser utilizada. Adaptações para uso de diferentes controladores são inviáveis ou somente realizadas a altos custos. Adicionalmente, a computação realizada pelos controladores também deve ser temporalmente previsível, de modo a satisfazer os requisitos demandados pelo ambiente.

- Supervisão: a monitoração dos diversos processos e dispositivos de uma planta industrial é realizada através dos sistemas de supervisão ou sistemas supervisórios. Nesses sistemas, pode-se verificar o estado atual de um determinado sensor, válvula ou motor; realizar pequenas modificações na planta tais como a mudança de um *setpoint* ou a definição de alarmes de monitoramento; armazenar o funcionamento da planta para fins de histórico e registrar as atividades realizadas pelos usuários do sistema, dentre outras operações. Conforme apresentado na figura 1.1, os níveis inferiores disponibilizam serviços para que os níveis superiores executem ações em um nível de abstração maior. Dentre as tendências atuais dos sistemas supervisórios pode-se citar a inclusão de mecanismos para confiabilidade, previsibilidade e interoperabilidade e a integração com o ambiente web [102].

A integração, tanto horizontal (dentro de um mesmo nível) quanto vertical (entre níveis diferentes), dos sistemas industriais é um desafio que envolve padronizações, tecnologias e questões de mercado. O

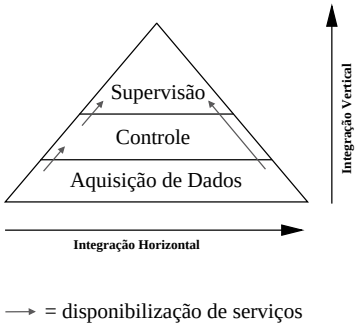


Figura 1.1: Níveis típicos de um sistema industrial de supervisão e controle

objetivo dessa integração é obter dados significativos sob o ponto de vista gerencial e administrativo, de modo a possibilitar a elaboração de metas, estimativa de perdas e análises de riscos.

Os sistemas de S&C constituem um dos sistemas industriais mais importantes e utilizados. Entretanto, outras classes de sistemas são também comumente encontrados nestes ambientes, a exemplo dos sistemas de comando numérico, sistemas visuais de inspeção de produtos e sistemas de transporte de materiais.

1.4 Motivação e objetivos

Justificado pela necessidade de disponibilização de plataformas de software que facilitem o desenvolvimento de sistemas modernos de S&C, este trabalho apresenta a concepção, projeto e implementação de uma arquitetura aberta para sistemas distribuídos de S&C, baseada na utilização de componentes de tempo-real. Essa plataforma, denominada ARCOS (Arquitetura para Controle e Supervisão), disponibiliza componentes e ferramentas básicas para as atividades de aquisição de dados, controle e supervisão. O ARCOS apresenta uma solução reutilizável de software que concentra conhecimentos de projeto e implementação acerca dos principais serviços necessários em sistemas de S&C e define pontos onde a plataforma pode ser especializada, de modo a ser utilizada em situações específicas de S&C. Através da reutilização dos componentes implementados no ARCOS, o desenvolvimento de novos sistemas de S&C é significativamente facilitado. O desenvolvedor conta com componentes pré-desenvolvidos que garantem um ambiente flexível, reutilizável, escalável, interoperável e com um bom grau de previsibilidade temporal.

A engenharia de software define o termo *framework* como um conjunto de classes que implementa de forma genérica e parcial os requisitos recorrentes em sistemas de um determinado domínio de aplicação [23]. Um *framework* é a implementação de uma solução arquitetural bem projetada e que pode ser reutilizada em situações onde o projetista não tem experiência suficiente, ou por desconhecimento do domínio da aplicação ou por incipiência no projeto de sistemas baseados em componentes, com o objetivo de adotar uma solução mais confiável e baseada em um projeto arquitetural mais maduro. Visto sob este

prisma, o ARCOS disponibiliza um *framework* que implementa os requisitos recorrentes em sistemas de S&C e define pontos de especialização, baseados em conexões de componentes, para a adequação desta solução para uso em situações específicas de aquisição, controle e supervisão. Como se sabe, componentes de software têm sido uma promissora tecnologia para o projeto e implementação de *frameworks* [23, 81]. A natureza flexível e modular desta abordagem é naturalmente compatível com as operações de especialização, visto que estas são freqüentemente realizadas através de componentes conectados à estrutura definida pelo *framework*.

Dentre as padronizações para objetos distribuídos, a especificação CORBA 2.x (*Common Object Request Broker Architecture* versões 2.x) [41] tem sido constantemente utilizada no desenvolvimento de sistemas caracterizados por uma heterogeneidade significativa em relação a linguagens de programação, sistemas operacionais, arquiteturas de hardware e canais de comunicação. Em particular, a versão 3 dessa especificação (CORBA 3) especifica um modelo de componentes distribuídos denominado CCM (*CORBA Component Model*) [71], caracterizado pela forte presença de mecanismos para conexão, configuração, empacotamento e implantação de componentes. As especificações CORBA e CCM foram criadas pelo *Object Management Group* (OMG) [68], um consórcio para produção e manutenção de padronizações para sistemas computacionais.

Interoperabilidade em sistemas distribuídos é a propriedade que permite a comunicação entre esses sistemas, a despeito da linguagem de programação, sistema operacional, meio de comunicação e arquitetura de hardware utilizados. Para que essa interoperabilidade seja efetiva, soluções abertas e padronizadas devem ser adotadas em todos os níveis do sub-sistema de comunicação utilizado. Assim como o protocolo TCP/IP garante a interoperabilidade nos níveis de roteamento e transporte de mensagens na Internet [16, 96], o CORBA define padronizações para especificação e implementação de objetos distribuídos e como estes podem ser acessados através de clientes. Entretanto, interoperabilidade plena é somente alcançada através do uso de padronizações para a camada de aplicação, onde a lógica do negócio está definida. Ao adotar-se uma especificação padronizada para um determinado domínio de aplicação, a integração com soluções desenvolvidas por outros fabricantes é consideravelmente facilitada, desde que estas soluções também estejam em conformidade com a padronização em questão. O OMG define algumas especificações, conhecidas como *CORBA Domains*, para domínio específicos de aplicações, tais como medicina, telefonia e controle de tráfego aéreo.

Dentre essas especificações, o DAIS (*Data Acquisition from Industrial Systems*) [69] define um serviço padronizado para aquisição de dados em sistemas industriais de S&C, considerando aspectos tais como a temporalidade dos dados e desempenho das atividades de coleta e entrega de dados. Por ser uma padronização CORBA, o DAIS usufrui de todas as potencialidades dessa tecnologia, incluindo a interoperabilidade efetiva e a possibilidade de utilização da tecnologia de componentes para a implementação desse serviço.

Previsibilidade temporal e tolerância a falhas[2] são requisitos fundamentais na grande maioria dos sistemas de S&C. Nos sistemas baseados em CLP's, esses requisitos são trivialmente satisfeitos através do uso de estruturas simples (e, conseqüentemente, previsíveis) de programação, como por exemplo a execução cíclica e mono-programada de um procedimento. Por outro lado, essas estruturas simples de programação constituem um entrave para o desenvolvimento de aplicações modernas de S&C, visto que os CLP's possuem restrições para processamento, armazenamento e comunicação.

Pesquisas recentes procuram disponibilizar ambientes de software que se beneficiam do uso da tecnologia de componentes e, ao mesmo tempo, garantem algum grau de previsibilidade temporal nas operações realizadas por esses componentes. Um desses esforços de pesquisa é o conjunto de soluções para tempo-real desenvolvidas pelo *Distributed Object Computing* (DOC) *Group* [37] da *University of California, Irvine* e *Washington University*, EUA. Dentre essas soluções, destacam-se o TAO (*The ACE ORB*) [91] e o CIAO (*Component-Integrated ACE ORB*) [103, 65]. O TAO é uma implementação da especificação CORBA 2.x com otimizações e extensões para uso em sistemas de tempo-real. O TAO tem sido amplamente utilizado em diversas situações onde deseja-se aliar os benefícios do CORBA com a previsibilidade temporal requerida pelos sistemas de tempo-real, a exemplo de sistemas de controle de aeronaves e sistemas de telefonia. Devido ao fato de ser uma implementação da versão 2.x do CORBA, o TAO não trabalha com o conceito de componentes de software. O CIAO é uma implementação do CCM, com o objetivo de disponibilizar uma plataforma funcional para desenvolvimento baseado em componentes, que garanta a execução temporalmente previsível dos componentes e dos serviços.

Neste trabalho são apresentados o projeto, implementação e validação de uma plataforma baseada em componentes CCM/CIAO e que disponibiliza soluções flexíveis, interoperáveis e reutilizáveis para sistemas distribuídos de S&C. O ARCOS disponibiliza uma implementação, baseada em componentes de tempo-real, da especificação DAIS, com o objetivo de garantir a interoperabilidade com outras ferramentas que seguem a padronização proposta por essa especificação. O ARCOS remodela a especificação DAIS, utilizando a tecnologia de componentes, sem entretanto gerar impactos na comunicação com clientes DAIS CORBA 2.x. Adicionalmente, a implementação do DAIS realizada pelo ARCOS prevê a especialização do *framework* para obtenção de dados a partir de tecnologias distintas de dispositivos, como por exemplo, CLP's ou porta paralela. O ARCOS define uma arquitetura flexível para utilização de estratégias de controle, facilitando modificações para uso de um tipo ou outro de controlador. Um componente para controle PID foi desenvolvido no ARCOS para exemplificar como essa especialização é realizada. Para as atividades de supervisão, o ARCOS disponibiliza duas ferramentas: o *DAIS Server Browser* (versões *desktop* e web) e o *DAIS Server Manager*. O *DAIS Server Browser* permite a comunicação com qualquer implementação de um serviço DAIS, permitindo a obtenção e monitoramento de dados industriais. O *DAIS Server Manager* reúne ferramentas para gerenciamento do servidor DAIS.

[2] Capacidade de um sistema funcionar corretamente mesmo na presença de um número limitado de falhas.

Para validação da plataforma foram realizados experimentos de supervisão de um simulador de reator químico, com o objetivo de testar a flexibilidade e os serviços de supervisão disponibilizados pela implementação. Uma simulação de controle de velocidade de um veículo valida a implementação dos componentes de controle.

1.5 Estrutura da dissertação

Esta dissertação encontra-se dividida em cinco capítulos e sete apêndices.

O capítulo 2 apresenta os fundamentos básicos sobre componentes e sua aplicação em sistemas de tempo-real. São apresentados o modelo de componentes CCM proposto pelo OMG, suas facilidades para conexão e considerações sobre o processo de montagem e implantação de sistemas baseados em componentes. O capítulo 2 discute também os principais serviços disponibilizados pelo TAO e pelo CIAO. Em função do número reduzido de documentos sobre o TAO e o CCM/CIAO, nesse capítulo procurou-se elaborar um texto auto-contido e suficiente, do modo a constituir uma boa referência para futuros utilizadores do CCM/CIAO e do ARCOS.

O capítulo 3 apresenta a arquitetura de supervisão e controle proposta (ARCOS), justificando as decisões de projeto e tecnologias utilizadas. Aspectos de implementação são discutidos através do levantamento de recursos do TAO/CIAO que possibilitaram a implementação do *framework*. O capítulo 3 apresenta ainda qual a estratégia utilizada para a disponibilização de uma plataforma preocupada com a previsibilidade temporal das execuções.

O capítulo 4 apresenta como o *framework* arquitetural pode ser especializado para uso em uma situação particular de S&C. São apresentadas as formas de especialização dos componentes de aquisição de dados e controle, bem como as configurações necessárias para conectar os componentes. São apresentados ainda os experimentos de supervisão e controle desenvolvidos sobre a plataforma. Um experimento de supervisão de um simulador de reator químico ilustra como o DAIS pode ser utilizado para a aquisição e monitoração de dados industriais. Uma simulação de controle de velocidade de um veículo ilustra como os componentes de controle são utilizados e como ocorre a sua integração com os componentes de aquisição de dados. O capítulo 4 apresenta ainda uma avaliação da plataforma, bem como os trabalhos correlatos.

O capítulo 5 apresenta as conclusões da dissertação, indicações de trabalhos futuros e as publicações originadas deste trabalho.

O apêndice A apresenta a estrutura do descritor XML utilizado para a implantações de aplicações baseadas no CIAO e o objetivos de cada uma das *tags* presentes. O apêndice B apresenta o formato do arquivo de configuração da ferramenta de gerenciamento da compilação do CIAO e do ARCOS. O apêndice C apresenta um roteiro sobre como construir, compilar e implantar um componente utilizando o CIAO. O apêndice D apresenta os passos necessários para a compilação e instalação do CIAO e do

ARCOS. O apêndice E contém os arquivos IDL dos componentes da plataforma ARCOS. O apêndice F apresenta o programa LADDER utilizado para a supervisão do simulador de reator químico. O apêndice G apresenta o manual de utilização da ferramentas disponibilizadas pelo ARCOS.

Capítulo 2

Plataformas e Serviços de Tempo-Real

À medida em que a complexidade dos sistemas computacionais cresce, a utilização de plataformas de software que facilitam o desenvolvimento se torna uma prática fundamental para o sucesso dessas aplicações. Pode-se distinguir dois tipos de serviços presentes em tais sistemas: os serviços funcionais e os serviços não-funcionais. Os serviços funcionais implementam a lógica de negócio da aplicação e são particulares para o sistema em questão. No caso dos sistemas de S&C, este serviços devem garantir a corretude lógica e temporal da aplicação. Os serviços não-funcionais são responsáveis por atividades inerentes do ambiente computacional no qual o sistema é executado, tais como distribuição, segurança, persistência de dados, controle de transações, tolerância a falhas e adaptação. Esses serviços, pelo fato de serem recorrentes em uma variedade de sistemas, são candidatos potenciais à reutilização. O objetivo, ao utilizar uma plataforma de tempo-real, é permitir que o desenvolvedor dedique a maioria dos seus esforços na implementação dos serviços funcionais. Nessa abordagem, os serviços não-funcionais são disponibilizados pela plataforma de software e configurados a partir de arquivos descritores ou ferramentas.

Este capítulo apresenta os conceitos e mecanismos utilizados no projeto e desenvolvimento do ARCOS. São apresentados o modelo de componentes do CORBA (CCM), as funcionalidades e serviços disponibilizados pelo TAO/CIAO e a padronização para aquisição de dados especificada pelo DAIS.

2.1 Middleware e sistemas de tempo-real

Middleware certamente é um termo muito utilizado ao longo da história da computação. Nota-se que, desde o final da década de 80 até os dias atuais, a idéia de *middleware* passou por algumas mudanças, de modo a refletir os novos objetivos e idéias que surgiam [6]. De um modo geral, o *middleware* desempenha um papel de facilitador do processo de desenvolvimento, situando-se entre o sistema operacional e as aplicações. O objetivo é dispensar o desenvolvedor de tarefas enfadonhas e propensas a erros, aumentando a produtividade e a confiabilidade do sistema desenvolvido.

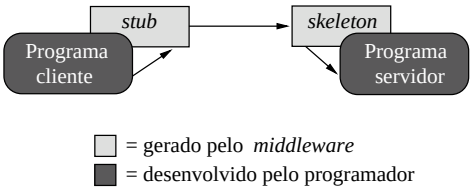


Figura 2.1: Comunicação de processos através do RPC

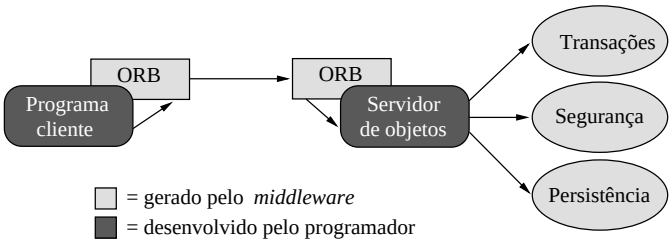


Figura 2.2: Comunicação de processos e utilização de serviços através do ORB

As primeiras soluções de *middleware* para sistemas distribuídos tinham como objetivo a realização de atividades básicas ligadas ao desenvolvimento desses sistemas. O RPC (*Remote Procedure Call*) [7, 63] é um exemplo de *middleware* que poupa o desenvolvedor da implementação de código para comunicação entre processos numa rede. Esse código, recorrente e propenso a erros se construído por programadores inexperientes, é automaticamente gerado pelo *middleware* e encapsulado em unidades denominadas *stubs* e *skeletons*, conforme ilustrado na figura 2.1. O desenvolvedor se limita ao desenvolvimento dos serviços funcionais da aplicação, estando estes localizados numa só máquina ou distribuídos numa rede de computadores. Essa facilidade é geralmente conhecida como transparência de localização.

Uma evolução natural dessa abordagem foi o surgimento dos ORB's (*Object Request Broker*), o que propiciou a utilização de recursos da orientação a objetos na comunicação entre processos distribuídos. Assim como as funcionalidades para computação distribuída, outros serviços tais como persistência, segurança, controle de transações, balanceamento de carga e comunicação baseada em eventos passaram a fazer parte da solução proposta pelo *middleware*. Conforme ilustrado na figura 2.2, o *middleware* baseado em objetos realiza o gerenciamento da comunicação distribuída e disponibiliza serviços não-funcionais que podem ser utilizados pelas aplicações, com o objetivo de prover a funcionalidade requerida.

2.1.1 CORBA (Common Object Request Broker Architecture)

Diversas especificações de *middleware* baseados em objetos estão disponíveis atualmente, destacando-se o CORBA 2.x [41], o EJB (*Enterprise Java Beans*) [61] e o DCOM (*Distributed COMponents*) [60]. O CORBA 2.x é uma especificação para computação distribuída baseada em objetos, desenvolvida desde 1991 pelo OMG. Um dos objetivos dessa especificação é promover a comunicação distribuída entre objetos, a despeito da linguagem de programação na qual foram desenvolvidos e do ambiente utilizado para

executá-los. Desta forma, o CORBA 2.x é uma solução que promove independência de linguagem de programação, sistema operacional, arquitetura de hardware e rede de comunicação utilizados; constituindo uma abordagem adequada para uso em sistemas de S&C, visto que estes são geralmente caracterizados por uma heterogeneidade aparente.

A figura 2.3 apresenta os principais elementos da arquitetura CORBA 2.x, descritos a seguir.

- *Stubs* e *Skeletons*: de forma semelhante ao RPC, os *stubs* e *skeletons* são peças de código geradas pelo *middleware* e realizam atividades de empacotamento e desempacotamento (*marshalling* e *unmarshalling*) de parâmetros e comunicação com as camadas subjacentes da arquitetura. Os *stubs* e *skeletons* são gerados a partir da compilação de especificações independentes de linguagem de programação, representadas por arquivos IDL (*Interface Definition Language*). O compilador de IDL's é disponibilizado pela implementação CORBA 2.x e gera *stubs* e *skeletons* para uma linguagem de programação particular.

- ORB e GIOP: o ORB é o elemento responsável pelo gerenciamento de referências remotas e comunicação com outros ORB's e utiliza uma especificação chamada GIOP (*General Inter-ORB Protocol*) para definir como enviar requisições e receber respostas de outros ORB's. O GIOP é um protocolo genérico que garante a interoperabilidade em ambientes com tecnologias distintas de comunicação. Outras especificações realizam esse trabalho em ambientes de rede específicos, a exemplo do IIOP (*Internet Inter-ORB Protocol*) que é uma especialização do GIOP para uso em redes baseadas nos protocolos TCP/IP.

- Adaptador de Objetos: o adaptador de objetos é o elemento responsável pela gerência do comportamento dos objetos distribuídos. Através dele é possível definir políticas relacionadas ao ciclo de vida, persistência e escalabilidade destes objetos. Em um mesmo sistema podem existir diversos adaptadores de objetos, cada um deles configurado com políticas que provêm diferentes comportamentos.

- *Servant*: o CORBA 2.x realiza uma distinção importante, porém sutil, entre objetos e *servants*. Um objeto define uma referência remota para uma entidade que disponibiliza serviços. Invocações realizadas a partir dessa referência são executadas por um *servant* que é "encarnado" naquele objeto. De modo a garantir aspectos tais como escalabilidade, a correspondência entre objetos e *servants* pode não ser do tipo um-para-um. Antes de receber invocações remotas, um *servant* precisa ser **ativado** em algum adaptador de objetos. Após essa ativação, o *servant* passa a ter seu comportamento definido pelas políticas do adaptador de objetos no qual ele foi ativado. O *servant* é um elemento que deve ser explicitamente codificado pelo desenvolvedor.

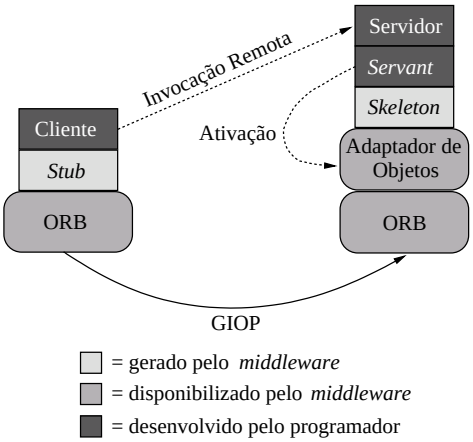


Figura 2.3: Principais elementos da arquitetura CORBA 2.x

- Cliente e Servidor: o cliente e o servidor são aplicações construídas pelo desenvolvedor. O servidor realiza as atividades de inicialização do ORB, criação e configuração de adaptadores de objetos; criação e ativação de *servants*, dentre outras. O desenvolvimento de servidores CORBA 2.x confiáveis, escaláveis e reutilizáveis é uma tarefa custosa e que exige experiência por parte do desenvolvedor. O cliente é uma aplicação que adquire referências remotas para objetos e as utiliza para invocar operações realizadas pelo servidor.

O CORBA 2.x, apesar das vantagens relacionadas ao suporte à heterogeneidade, apresenta algumas deficiências que dificultam o seu uso potencial no desenvolvimento de sistemas distribuídos complexos:

- Ausência de mecanismos para a conexão desacoplada de objetos: objetos cuja implementação dependem de outros objetos precisam localizar estes objetos e conectar-se a eles de forma explícita, através de código implementado pelo desenvolvedor. Essa abordagem conduz a sistemas inflexíveis e com baixo grau de reutilização.
- Ausência de uma padronização para servidores: o CORBA 2.x não disponibiliza um *framework* genérico para atividades recorrentes de servidores, tais como: inicialização do servidor, configuração de serviços (localização, eventos, etc) e gerenciamento do ambiente de execução (políticas do adaptador) de cada objeto. Os servidores precisam ser inteiramente codificados pelo desenvolvedor, exigindo conhecimento aprofundado da arquitetura CORBA 2.x e demandando esforços que não estão relacionados à parte funcional do negócio.
- Ausência de padronizações para configuração e implantação: o CORBA 2.x não define mecanismos para implantação e configuração remota de objetos. As operações de implantação, configuração e inicialização são realizadas de forma *ad-hoc*, em cada máquina que compõe o sistema distribuído, dificultando o gerenciamento e manutenção de sistemas complexos.

Outras deficiências do CORBA 2.x constituem um impeditivo para a sua utilização em sistemas críticos em relação ao tempo:

- Ausência de interfaces para especificação de QoS (*Quality of Service*): o CORBA 2.x não disponibiliza interfaces para especificação de parâmetros de QoS. Aplicações CORBA 2.x não conhecem informações sobre prioridades, taxas de invocações de serviços ou políticas de controle de admissão de novos clientes.
- Ausência de mecanismos para garantia de QoS: a maioria das implementações CORBA realiza a transmissão e execução de invocações seguindo a estratégia FIFO (*First-In First-Out*), podendo levar à ocorrência de inversões de prioridade [80, 49]. Adicionalmente, o CORBA 2.x não permite que prioridades sejam atribuídas às *threads* que processam as requisições e a alocação de recursos, como memória, é feita de forma *ad-hoc* e sem uma preocupação com um possível consumo excessivo.
- Ausência de otimizações de desempenho: as implementações convencionais do CORBA 2.x são caracterizadas por um alto *overhead*, além de diversos pontos que implicam numa imprevisibilidade de execução. Essa imprevisibilidade é geralmente causada por: cópia excessiva de dados, utilização de *buffers* com comportamento não-uniforme, algoritmos imprevisíveis para demultiplexação e *dispatching*, cadeias longas de invocações a métodos virtuais e ausência de integração com as camadas subjacentes de sistema operacional e rede [91].

Frente a essas questões, padronizações que suportassem os requisitos para a aplicação do CORBA em sistemas de tempo-real passaram a ser desenvolvidas, a exemplo do RT-CORBA (*Real-Time CORBA*) [73] e do FT-CORBA (*Fault-Tolerant CORBA*) [74].

2.2 Componentes e sistemas de tempo-real

O desenvolvimento das abstrações e mecanismos que compõem a orientação a objetos foi um dos grandes avanços que possibilitaram um maior gerenciamento da crescente complexidade dos sistemas computacionais. Apesar de uma maior modularização e reutilização promovida pela orientação a objetos, linguagens de programação OO (Orientadas a Objetos) introduziam suas próprias derivações e extensões do paradigma, dificultando a interoperabilidade. Além disso, a inexistência de padronizações para representação de objetos e invocações de métodos tornava essas implementações específicas para uma determinada linguagem de programação e ambiente de execução.

A utilização moderada de recursos computacionais, bem como o gerenciamento das relações recursos/usuários são pontos críticos para o desenvolvimento de sistemas escaláveis. Através das operações de criação (instanciação) e remoção de objetos, o desenvolvedor deve alocar e desalocar os recursos necessários para a execução de determinada atividade. A implementação inexperiente ou não planejada dessas operações pode acarretar em sistemas inflexíveis e com escalabilidade limitada.

Sistemas distribuídos convencionais são, em sua maioria, caracterizados pela necessidade de serviços não-funcionais, tais como: localização de objetos, persistência, segurança, controle de transações, tolerância a falhas, dentre outros. Mesmo com a disponibilização desses serviços sob a forma de bibliotecas de objetos, a integração entre a aplicação sendo desenvolvida e estas bibliotecas é realizada explicitamente em código, demandando tempo e conhecimento especializado. Além disso, mudanças nas tecnologias utilizadas nesses serviços implicam numa modificação e recompilação da aplicação.

Em aplicações complexas é comum objetos se relacionarem com outros objetos de modo a implementar uma funcionalidade em particular. Esses relacionamentos costumam ser implementados explicitamente em código, produzindo objetos com reutilização restrita e sistemas onde a substituição de uma implementação de objeto por outra é geralmente realizada a altos custos.

Uma abordagem que procura superar estas deficiências é a tecnologia de componentes de software. Duas definições de componentes de software encontradas na literatura são:

> *"Um componente é um elemento de software, em conformidade com um modelo, que pode ser independentemente implantado e conectado a outros componentes, sem a necessidade de modificações."*
>
> Heineman, 2001 [45]

> *"Componentes reutilizáveis de software são artefatos auto-contidos e facilmente identificáveis que descrevem e/ou implementam funções específicas e possuem interfaces bem definidas, documentação apropriada e potencialidade para reutilização."*
>
> Sametinger, 2001 [83]

O principal objetivo da engenharia de software baseada em componentes é desenvolver sistemas a partir da conexão facilitada de peças pré-construídas de software (componentes). A tecnologia de componentes supera as deficiências da orientação a objetos a partir da disponibilização dos seguintes mecanismos [92]:

- Definição de padronizações para comunicação: com essa nova abordagem, um componente passa a ser uma entidade em conformidade com padronizações que descrevem o formato de representação de componentes e a forma de realização de invocações. Cada componente possui interfaces que descrevem os serviços disponibilizados e pontos bem definidos para conexão com outros componentes.
- *Container*: componentes são executados em um ambiente pré-disponível de software denominado

container. Após a **implantação** (*deployment*) de um componente em um *container*, este passa a gerenciar atividades importantes na execução daquele. O gerenciamento do ciclo de vida do componente, bem como a sua relação com os possíveis clientes é exclusivamente realizada pelo *container*, numa operação geralmente chamada de *instance pooling*. O objetivo é gerenciar os recursos disponíveis (componentes) de modo a garantir uma boa escalabilidade da aplicação.

- Utilização facilitada de serviços não-funcionais: além das facilidades acima descritas, a integração do componente com os serviços não-funcionais é também gerenciada pelo *container*, com a ajuda de indicações simplificadas geralmente realizadas através de arquivos XML construídos pelo desenvolvedor (descritores de implantação). Todas essas operações contribuem significativamente para a produtividade e qualidade das soluções, visto que o desenvolvedor se preocupa somente com o desenvolvimento de serviços funcionais (lógica da aplicação).

- Mecanismos flexíveis para conexão de componentes: na abordagem baseada em componentes, sistemas são implementados através da conexão de componentes pré-construídos. Uma determinada configuração, indicando os componentes participantes e as conexões entre eles é geralmente chamada de **montagem** (*assembly*). Durante o processo de implantação, todas as operações de conexão entre componentes é realizada pelo *container*, com a ajuda de indicações simplificadas presentes nos arquivos descritores de implantação. A existência de ferramentas geradoras de arquivos descritores de implantação, a partir de configurações realizadas pelo desenvolvedor de forma visual, representam mais um avanço em direção ao desenvolvimento produtivo de sistemas confiáveis. Adicionalmente, essa montagem pode ser modificada de modo a atender necessidades de mudanças do sistema. Tem-se, desta forma, um novo paradigma para manutenção de sistemas, podendo esta inclusive ser realizada com o sistema em funcionamento. As conexões entre componentes seriam modificadas de modo a permitir a substituição de um componente por outro.

A tabela 2.1 resume as principais diferenças entre as tecnologias de objetos e componentes. Enquanto a orientação a objetos define mecanismos para o desenvolvimento baseado em abstrações (encapsulamento, ocultamentos e herança), a tecnologia de componentes se preocupa com a definição de mecanismos para a composição facilitada de soluções. Outra diferença importante é em relação ao gerenciamento do ciclo de vida. Enquanto objetos têm seu ciclo de vida gerenciado diretamente pelo desenvolvedor (através de operações do tipo *new* - criação de instância e *delete* - destruição de instância), os componentes são geralmente executados em um ambiente de software pré-construído (*container*). Esse *container* gerencia o processo de criação e destruição de novas instâncias de componentes, definindo políticas escaláveis de utilização desses recursos. A combinação de objetos, com o objetivo de construir um sistema completo, é geralmente realizada através de código-fonte, constituindo uma abordagem não tão flexível e que requer

OBJETOS	COMPONENTES
Ênfase nas abstrações	Ênfase na composição (*assembly*)
Têm seu ciclo de vida gerenciado pelo desenvolvedor	Têm seu ciclo de vida gerenciado por um ambiente de software pré-construído (*container*)
Composição realizada via código-fonte	Composição realizada através de mecanismos definidos pela especificação
São unidades de instanciação	São unidades de implantação (*deployment*)

Tabela 2.1: Diferenças conceituais entre objetos e componentes

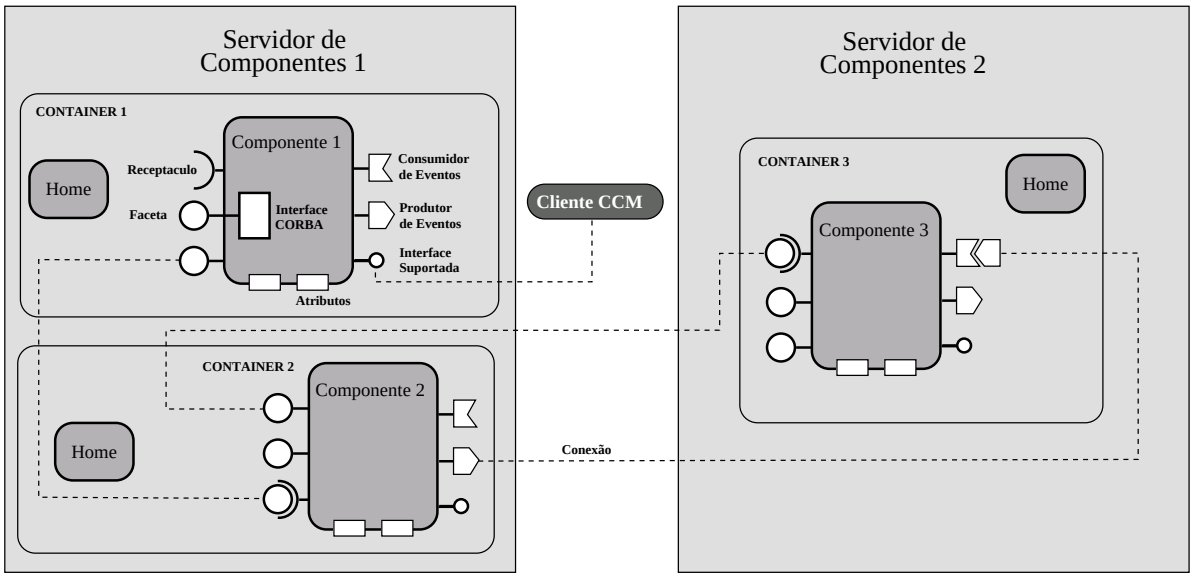


Figura 2.4: Modelo de componentes do CCM

recompilações para a alteração de alguma dessas combinações. A tecnologia de componentes prevê a definição de mecanismos padronizados para conexão, e a utilização de tecnologias flexíveis para a especificação dessas combinações como, por exemplo, arquivos descritores XML. Diversos modelos e padronizações de componentes têm sido propostos em pesquisas recentes, com destaque para as padronizações CORBA Component Model (CCM) [71], EJB (*Enterprise Java Beans*) [61] e DCOM (*Distributed COMponents*) [60].

2.2.1 CCM (CORBA Component Model)

Motivado pelas deficiências já citadas da arquitetura CORBA 2.x, o OMG disponibilizou em junho de 2002 a versão 3 do CORBA, contemplando um modelo e uma especificação de componentes denominada *CORBA Component Model* (CCM). O CCM especifica um modelo de componentes distribuídos, mecanismos para conexão de componentes e padronizações para implantação, configuração e manutenção de

componentes. A figura 2.4 apresenta o modelo de componentes definido pelo CCM.

2.2.1.1 O modelo de componentes

Um componente CCM é uma unidade de implantação e conexão que implementa funcionalidades reutilizáveis do sistema computacional em questão. Essas funcionalidades são disponibilizadas através de métodos pertencentes à(s) **interface(s) suportada(s)** pelo componente. Uma interface suportada é uma interface CORBA da qual o componente realiza uma implementação. Essa interface define os métodos que podem ser invocados por aplicações clientes daquele componente.

As conexões entre componentes são realizadas através de mecanismos denominado ***ports***. O CCM define quatro tipos de *ports*:

- Facetas (*facets*): uma faceta representa uma funcionalidade disponibilizada pelo componente para fins de conexão. Uma determinada faceta de um componente é identificada através de um nome e de um tipo, sendo este tipo definido através de uma interface CORBA.

- Receptáculos (*receptacle*): um receptáculo representa uma funcionalidade requerida pelo componente e, em conjunto com as facetas, representam um mecanismo para conexão entre componentes. Por exemplo, um sistema distribuído de uma grande organização poderia ser formado pelo componente *FinancialReport*, o qual possui um receptáculo conectado à faceta do componente *SortingAlgorithm*. Para realizar a ordenação do relatório, o componente *FinancialReport* invoca, através do seu receptáculo, os métodos definidos na interface utilizada na declaração da faceta do componente *SortingAlgorithm*. O CCM define dois tipos de receptáculos: *simplex*, aqueles que podem ser conectados a no máximo uma faceta; e *multiplex*, aqueles que podem ser conectados a mais de uma faceta.

- Produtores de Eventos (*event source*): as invocações, via receptáculos, de métodos pertencentes à faceta conectada é feita de forma síncrona (bloqueante), ou seja, o componente que possui o receptáculo fica bloqueado até que o componente que possui a faceta execute todo o método solicitado. Este comportamento é inadequado em situações onde uma comunicação assíncrona, desacoplada e do tipo muitos-para-muitos é necessária. Os produtores de eventos publicam uma determinada informação e esta é entregue, assincronamente, a todos os componentes interessados, caracterizando um modelo *publisher/subscriber* de comunicação. O CCM define dois tipos de produtores de eventos: aqueles com os quais um evento só pode ser enviado para um depósito e aqueles com os quais um evento pode ser enviado para vários depósitos.

- Depósitos de Eventos (*event sink*): componentes podem definir depósitos de eventos e conectá-los apropriadamente a produtores de eventos de outros componentes. No exemplo acima citado,

suponha que o componente *ReportManager* realize a solicitação de relatórios ao componente *FinancialReport*. Em virtude da complexidade e quantidade de dados envolvidos na geração desses relatórios, é indesejável que o componente *ReportManager* permaneça bloqueado durante esta operação. Define-se, portanto, um produtor de eventos no componente *ReportManager* e um depósito de eventos no componente *FinancialReport*. Estes eventos indicam, assincronamente, que o *ReportManager* solicita um relatório ao *FinancialReport*. Adicionalmente, pode-se criar um produtor de eventos no componente *FinancialReport* e um depósito de eventos no componente *ReportManager*. Esses outros eventos indicam, assincronamente, que o relatório solicitado está disponível.

Vale salientar que as operações de conexão de facetas e receptáculos, bem como de produtores e depósitos de eventos, são realizadas pelo *container*, a partir de indicações nos arquivos descritores de implantação.

Além dos *ports*, o modelo de componentes do CCM define pontos de configuração do componente denominados **atributos**. Esses atributos, de forma semelhante às facetas e receptáculos, possuem um nome e um tipo. O valor do atributo é informado, via arquivo descritor de implantação, no momento da implantação do componente. Um atributo serve para adequar o funcionamento do componente a uma situação particular. Por exemplo, pode-se utilizar atributos para definir a orientação (retrato ou paisagem) e o número de registros por página do relatório gerado pelo componente *FinancialReport*. O CCM define ainda as chamadas **API's de navegação**, que possibilitam obter-se uma referência para o componente a partir de uma referência para uma faceta daquele componente e vice-versa. Cada componente CCM está associado a um outro componente denominado ***home***. Esse componente é responsável pelas atividades de criação de novos componentes e pela busca por componentes já existentes. Essas atividades de criação e busca são implementadas por métodos denominados *factories* e *finders*, respectivamente.

Cada componente CCM é executado em um ambiente de software padronizado chamado *container*. Para cada componente implantado haverá um *container* responsável pela sua execução. Uma coleção de *containers*, executando numa mesma máquina, compõe um servidor de componentes. Facetas e receptáculos, bem como produtores e depósitos de eventos, podem ser remotamente conectados, constituindo um bom mecanismo para o desenvolvimento de sistemas distribuídos flexíveis e escaláveis.

2.2.1.2 Arquivos IDL e CIDL

Devido ao fato de ser uma padronização baseada no CORBA, a definição de componentes CCM é realizada através de arquivos independentes de linguagem de programação, denominados arquivos IDL (*Interface Definition Language*) e CIDL (*Component Implementation Definition Language*).

Os arquivos IDL têm a função de definir componentes, suas interfaces suportadas e os *ports* requeridos e disponibilizados. Os arquivos CIDL, por sua vez, definem padrões de interação do componente com o

container que o executa.

Código 2.1: Exemplo de arquivo IDL

```
1  interface ISortingAlgorithm {
2     string get_description();
3  };
4  interface ISorting {
5     void sort(inout SortableData sortable_data);
6  };
7  component SortingAlgorithm supports ISortingAlgorithm {
8     provides ISorting sorting_facet;
9  };
10 home SortingAlgorithmHome manages SortingAlgorithm {};
11
12 eventtype doReport {};
13 eventtype reportDone {
14    public long milliseconds;
15 }
16
17 component FinancialReport {
18    uses ISorting sorting_receptacle;
19    publishes reportDone done;
20    consumes doReport do;
21 };
22 home FinancialReportHome manages FinancialReport {};
23
24 component ReportManager {
25    publishes doReport do;
26    consumes reportDone done;
27    attribute string orientation;
28 };
29 home ReportManagerHome manages ReportManager {};
```

O código 2.1 exemplifica os principais construtores em arquivos IDL. Conforme apresentado, facetas são implementadas através de interfaces que diferem das interfaces suportadas pelo componente. As linhas 1 a 3 definem a interface suportada pelo componente, ou seja, o conjunto de métodos acessíveis a partir da referência para o componente. O método *get_description()* retornaria, por exemplo, o nome do algoritmo implementado (*Bubble*, *Hash*, *Quick*). As linhas 4 a 6 definem a interface utilizada na declaração da faceta do componente *SortingAlgorithm*. O método *sort()* seria invocado pelo receptáculo conectado nesta faceta.

As linhas 7 a 10 definem o componente *SortingAlgorithm* e o seu *home*. A palavra reservada *provides* define uma faceta no componente em questão. Essa faceta é identificada por um tipo (*ISorting*) e por um nome (*sorting_facet*). As palavras reservadas *home* e *manager* indicam, respectivamente, o componente *home* e o componente de negócio gerenciado por este *home*. Visto que facetas, receptáculos, produtores e depósitos de eventos são estruturas tipadas[1], a conexão entre componentes deve ser realizada através de *ports* do mesmo tipo, ou com tipos que se relacionam através de herança e polimorfismo.

As linhas 12 a 15 definem os eventos utilizados para a solicitação e indicação de término dos relatórios. A linhas 17 a 22 definem o componente *FinancialReport*. A palavra reservada *uses* define um receptáculo no componente em questão. Esse receptáculo é também identificado por um tipo (*ISorting*) e um nome

[1] Termo utilizado em linguagens de programação para indicar que as variáveis são categorizadas. Exemplos de categorias: tipos primitivos (*int*, *float*, *double* etc) e classes.

(*sorting_receptacle*). A palavra reservada *publishes* indica que aquele componente possui um produtor de eventos do tipo *reportDone,* denominado *done* e que pode enviar eventos para vários depósitos. Se deseja-se restringir esse evento a somente um depósito usa-se a palavra reservada *emits*. A palavra reservada *consumes* indica que aquele componente possui um depósito de eventos do tipo *doReport* e denominado *do*.

Finalmente, as linhas 24 a 29 definem o componente *ReportManager*. A palavra reservada *attribute* define que aquele componente possui um atributo de configuração do tipo *string* e denominado *orientation* (retrato/paisagem). O valor desse atributo é informado, geralmente, através de arquivos descritores de implantação. Caso o atributo seja utilizado somente para leitura, utiliza-se a palavra reservada *readonly attribute*.

Código 2.2: Exemplo de arquivo CIDL

```
1  composition service SortingAlgorithm_Impl {
2     home executor SortingAlgorithmHome_Exec {
3        implements SortingAlgorithmHome;
4        manages SortingAlgorithm_Exec;
5     };
6  };
7
8  composition service FinancialReport_Impl {
9     home executor FinancialReportHome_Exec {
10       implements FinancialReportHome;
11       manages FinancialReport_Exec;
12    };
13 };
14
15 composition session ReportManager_Impl {
16    home executor ReportManagerHome_Exec {
17       implements ReportManagerHome;
18       manages ReportManager_Exec;
19    };
20 };
```

O código 2.2 exemplifica os principais construtores em arquivos CIDL. Nesse arquivo, indica-se que tipo de *container* irá hospedar cada componente. A palavra reservada *session* informa a categoria do componente em relação ao seu ciclo de vida. A tabela 2.2 resume as categorias especificadas pelo CCM:

- *Service*: utilizada em componentes que não necessitam de retenção de estado entre invocações de métodos. Esses componentes não são unicamente identificáveis e são úteis para modelar execuções auto-contidas e independentes.
- *Session*: utilizada em componentes que necessitam de retenção de estado entre invocações de métodos. Por exemplo, podem ser utilizados para modelar a sessão de um cliente em um sistema de informação.
- *Process*: utilizada quando necessita-se de componentes com estado persistente, porém os meios para identificação única desse componente são definidos pelo desenvolvedor.

Modelo de Componente	Chave Primária ?	Categoria do Componente
Stateless	Não	*Service*
Stateful	Não	*Session*
Persistente	Não	*Process*
Persistente	Sim	*Entity*

Tabela 2.2: Categorias de componentes em relação ao seu ciclo de vida

- *Entity*: utilizada quando necessita-se de componentes com estado persistente e deseja-se que o *container* disponibilize operações para identificação única desses componentes. Essa identificação única é alcançada através da utilização de um atributo da classe como chave primária e o valor desse atributo deve ser encontrado somente no componente que ele unicamente identifica.

No exemplo apresentado, os componentes *SortingAlgorithm* e *FinancialReport* são do tipo *stateless*, ao passo que o componente *ReportManager* é do tipo *stateful*. No CCM, as implementações de componentes são denominadas **executores**, numa abordagem semelhante aos *servants*, definidos pelo CORBA 2.x. No arquivo CIDL define-se os executores para os componentes *home*, indicando qual *home* eles implementam e qual componente de negócio eles gerenciam (linhas 2 a 5). Os executores para componentes *home* são automaticamente gerados pelas ferramentas de compilação de arquivos IDL e CIDL.

2.2.1.3 Descritores de implantação

Grande parte dos sistemas computacionais são caracterizados pela utilização de serviços não-funcionais tais como persistência de dados, localização de objetos, gerenciamento de transações distribuídas e tolerância a falhas, dentre outros. Devido à recorrência desses requisitos em grande parte dos sistemas, é conveniente que esses serviços passem a ser disponibilizados pelo *middleware*, poupando o desenvolvedor de trabalhos complexos, sujeitos a erros e que não estão diretamente relacionados com o negócio da aplicação.

Com a tecnologia de componentes de software, o gerenciamento do uso desses serviços, pelos componentes, é realizado pelo *container*. A indicação de quais serviços serão utilizados por um determinado componente, bem como parâmetros de configuração do uso desse serviço são geralmente indicados em arquivos descritores de implantação. No momento da implantação do componente, esses arquivos são lidos pelo *container* e as configurações necessárias são automaticamente realizadas, de modo a prover o serviço requisitado.

A especificação CCM define alguns arquivos descritores de implantação, cada um deles responsável pela indicação de atributos relacionados à configuração do *container* ou à utilização de serviços não-funcionais. Esses arquivos, representados no formato XML, são os seguintes:

- *Component Property File Descriptor* (.cpf): responsável pela informação dos valores iniciais dos atributos de cada componente. Por exemplo, poderia-se utilizar esse arquivo para indicar que, nessa montagem, o atributo *orientation* do componente *ReportManager* seria inicializado com o valor *portrait.*

- *CORBA Component Descriptor* (.ccd): contém informações para a devida implantação do componente no *container*, tais como: serviços não-funcionais a serem utilizados, comportamento multitarefa e a categoria do componente.

- *Component Assembly Descriptor* (.cad): nesse arquivo informa-se quantos componentes, de cada tipo definido, serão instanciados e como estes serão conectados, de forma a compor a aplicação final.

- *Software Package Descriptor* (.csd): contém detalhes técnicos tais como compilador e linguagens utilizados, além de indicar a localização dos outros arquivos descritores.

Devido ao fato de a construção desses arquivos descritores ser uma atividade enfadonha e sujeita a erros, geralmente são disponibilizadas ferramentas que automatizam essa construção, a partir de indicações realizadas pelo usuário.

Recentemente, como parte das especificações para implantação e configuração de componentes, o OMG padronizou a utilização de um arquivo descritor único (*flattened*), contendo as informações principais para essas atividades. O código 2.3 apresenta os principais elementos desse arquivo descritor, utilizando os componentes do exemplo de geração do relatório financeiro. Uma descrição completa de todas as *tags* utilizadas nesse arquivo descritor pode ser encontrada no apêndice A.

O bloco *implementation*, nas linhas 8 a 13, define um novo tipo de componente. Esse tipo será utilizado posteriormente, no bloco *instance*, para a criação de instâncias de componentes desse tipo. Cada novo tipo declarado possui um identificador (*id*) escolhido pelo desenvolvedor, porém é comum a presença do sufixo mdd (*Monolithic Deployment Description*). Nesse bloco são informados um nome para o tipo e os nomes dos blocos de artefatos (a serem descritos adiante). Para cada tipo de componente utilizado na aplicação deve existir um bloco *implementation* correspondente.

O bloco *instance*, nas linhas 16 a 21, indica a criação de uma nova instância de componente. Cada nova instância declarada possui um identificador (*id*) escolhido pelo desenvolvedor, porém é comum a presença do sufixo idd (*Instance Deployment Description*). Nesse bloco são informados um nome para a instância, o nó no qual esta instância será criada (para implantações remotas) e o tipo da instância[2]. Para cada instância a ser criada na aplicação deve existir um bloco *instance* correspondente.

[2] O tipo da instância é indicado através da *tag* <*implementation*> e deve ser igual ao *id* de um tipo definido em algum bloco *implementation*.

Código 2.3: Exemplo de arquivo descritor de implantação

```
1  <Deployment:deploymentPlan
2  xmlns:Deployment="http://www.omg.org/Deployment"
3  xmlns:xmi="http://www.omg.org/XMI"
4  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5  xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
6      ...
7
8      <implementation id="ReportManager-mdd">
9          <name>ReportManager-mdd</name>
10         <source></source>
11         <artifact>ReportManager_exec</artifact>
12         <artifact>ReportManager_svnt</artifact>
13     </implementation>
14     ...
15
16     <instance id="ReportManager-idd">
17         <name>ReportManager-idd</name>
18         <node>MainNode</node>
19         <source></source>
20         <implementation>ReportManager-mdd</implementation>
21     </instance>
22     ...
23
24     <connection>
25         <name>financialreport_sortingalgorithm_connection</name>
26         <internalEndpoint>
27             <portName>sorting_facet</portName>
28             <kind>Facet</kind>
29             <instance>SortingAlgorithm-idd</instance>
30         </internalEndpoint>
31         <internalEndpoint>
32             <portName>sorting_receptacle</portName>
33             <kind>SimplexReceptacle</kind>
34             <instance>FinancialReport-idd</instance>
35         </internalEndpoint>
36     </connection>
37     ...
38
39 </Deployment:deploymentPlan>
```

O bloco *connection*, nas linhas 24 a 36, indica a conexão de uma faceta com um receptáculo ou de um produtor com um depósito de eventos. Cada conexão é definida por um nome (*tag name*) e por dois participantes, denominados *internalEndpoints*. Para cada *internalEndPoint* indica-se o nome do *port* (deve ser o nome de um dos *ports* definidos para o componente no arquivo IDL), o tipo (faceta, receptáculo, produtor ou depósito) e a instância que contém aquele *port* e que participará da conexão[3]. Para cada conexão presente na aplicação deve existir um bloco *connection* correspondente.

Ressalta-se que o uso desses descritores de implantação contribui de forma significativa para o desenvolvimento de aplicações modulares, flexíveis e reutilizáveis. Operações de criação, configuração e conexão de componentes, antes realizadas através de códigos escritos pelo desenvolvedor, são automaticamente realizadas pelo *container*, após a leitura desses arquivos descritores de implantação. A disponibilização de ferramentas para geração desses arquivos é um outro fator que contribui para a produtividade e qualidade de desenvolvimento de sistemas baseados em componentes.

[3] O nome da instância participante é indicado através da *tag* <*instance*> e deve ser igual ao *id* de uma instância definida em algum bloco *instance*.

2.3 Frameworks e sistemas de tempo-real

Nesta seção serão apresentados os conceitos, classificações e objetivos dos *frameworks*, bem como os aspectos básicos do ACE - um *framework* para desenvolvimento de sistemas distribuídos de alto desempenho.

2.3.1 Frameworks

À medida em que a complexidade dos sistemas computacionais cresce, o projeto de soluções de software corretas, modulares e flexíveis se torna um trabalho cada vez mais árduo. Percebe-se que, de um modo geral, essas soluções são comprometidas devido a falhas de projeto relacionadas com a arquitetura de sistema adotada. O desenvolvimento de arquiteturas satisfatórias requer diversas habilidades do projetista, desde um profundo conhecimento do domínio da aplicação até experiência e maturidade no projeto de sistemas orientados a objetos. Um *framework* é um mecanismo que garante a utilização de um bom projeto arquitetural, além de alavancar a produtividade no desenvolvimento de sistemas.

> *"Um framework orientado a objetos é um conjunto de classes abstratas e concretas que implementam, de forma parcial, requisitos recorrentes em sistemas de um determinado domínio de aplicação."*
>
> Fayad et. al., 1999 [23]

Um *framework* é, portanto, a implementação de uma arquitetura que define um conjunto de objetos e a forma como estes cooperam para atingir os objetivos de um determinado domínio de aplicação, capturando conhecimento especializado sobre características recorrentes daquele domínio e definindo pontos de extensão, denominados ***hot-spots*** ou pontos de variabilidade, que representam variações entre aplicações deste domínio. As decisões de projeto que permanecerão invariáveis em qualquer instância do framework são denominadas ***frozen-spots*** ou pontos de estabilidade. Com a adoção de um *framework*, além de obter-se uma solução arquitetural já validada, o trabalho do desenvolvedor se resume à especialização dos *hot-spots* e à conexão destas especializações à estrutura definida pelo *framework*.

Uma das características do uso de um *framework* é a ocorrência de **inversão de controle**. Ao construir um sistema sem a utilização de um *framework*, o desenvolvedor projeta e implementa a estrutura e o fluxo principal de aplicação, fazendo invocações a métodos de bibliotecas quando necessário. Nessa abordagem, a possibilidade de construção de estruturas inflexíveis e falhas é consideravelmente maior se realizada por um profissional inexperiente ou sem conhecimento suficiente do domínio da aplicação. Ao utilizar um *framework*, a responsabilidade de projeto e implementação da estrutura e fluxo principal da aplicação é delegada para o *framework*, caracterizando a inversão de controle. Espera-se que a estrutura definida pelo *framework* seja reutilizável o suficiente para ser aplicada a uma variedade de aplicações daquele domínio, mediante especializações, realizadas pelo desenvolvedor, de pontos bem definidos (*hot-*

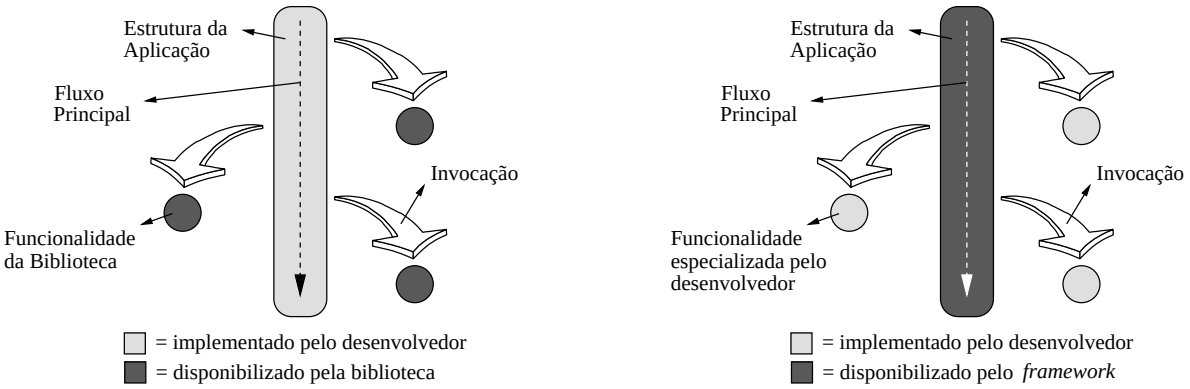


Figura 2.5: Desenvolvimento baseado em bibliotecas (à esquerda) e desenvolvimento baseado em *frameworks* (à direita)

spots). A figura 2.5 ilustra a diferença entre o desenvolvimento baseado em bibliotecas e o desenvolvimento baseado em *frameworks*.

Diversos *frameworks* têm sido propostos recentemente para aplicações em diversos domínios, entre eles: sistemas web, sistemas de processamento de sinais e sistemas distribuídos. Vale ressaltar que a análise do domínio no qual o *framework* está inserido é de fundamental importância para a sua real aplicação. *Frameworks* genéricos demais implicam em um esforço considerável, por parte do desenvolvedor, na especialização dos *hot-spots*. Por outro lado, uma especialização em demasia pode restringir consideravelmente o grupo de aplicações nas quais o *framework* pode ser utilizado.

O domínio de aplicação para o qual o *framework* foi desenvolvido pode ser tão amplo ou específico quanto possível. Por exemplo, pode-se projetar um *framework* para a construção de sistemas distribuídos, implicando numa grande variedade de aplicações-alvo, ou pode-se projetar um *framework* para aplicações multimídia, com um campo de atuação mais restrito. Adicionalmente, o *framework* para aplicações multimídia pode fazer uso do *framework* para sistemas distribuídos, enfatizando as características de reutilização e modularidade propostas por essa abordagem.

Os *frameworks* mais genéricos (que possuem uma maior variedade de aplicações-alvo) são chamados de horizontais, ao passo que os *frameworks* para domínios mais específicos são chamados de verticais [81]. Como exemplos de *frameworks* horizontais pode-se citar os utilizados no desenvolvimento de interfaces gráficas, nas operações de persistência em banco de dados e na implementações de políticas de segurança em aplicações. Como exemplos de *frameworks* verticais pode-se citar aqueles utilizados em sistemas de processamento de sinais, análises estatísticas ou sistemas industriais de supervisão e controle.

A figura 2.6 ilustra as três variáveis que podem ser utilizadas para distinguir os *frameworks* horizontais dos verticais:

- Generalidade: indica o número potencial de aplicações que podem utilizar recursos disponibilizados pelo *framework*.

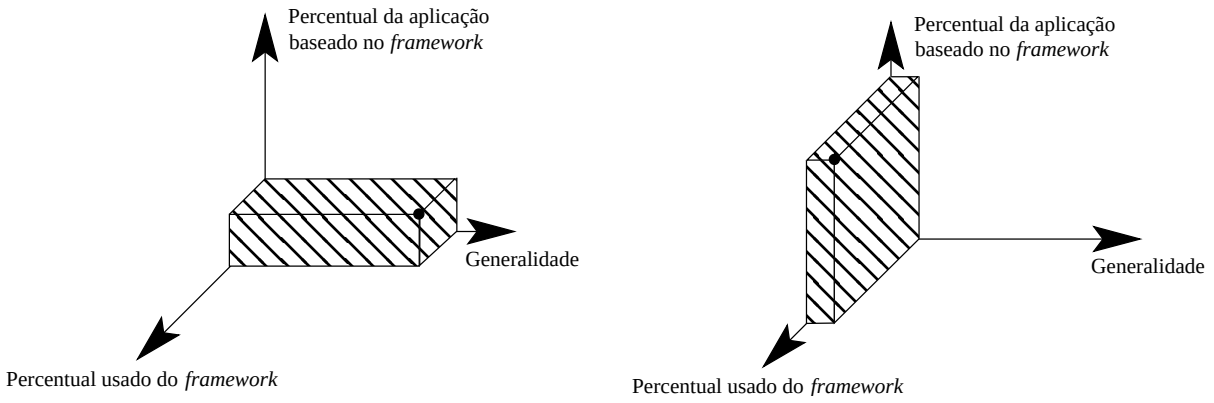


Figura 2.6: *Framework* horizontal (à esquerda) e *framework* vertical (à direita)
Fonte: [81]

- Percentual usado do *framework*: indica quanto do *framework* é utilizado em média pelas aplicações. Pode ser representado pela porcentagem de classes utilizadas e porcentagem de uso dos métodos das classes.
- Percentual da aplicação baseado no *framework*: indica quanto da aplicação foi construída baseando-se nos recursos disponibilizados pelo *framework*.

Quanto mais vertical for um *framework* menos genérico ele será, porém será melhor aproveitado e disponibilizará soluções mais completas. Em contrapartida, quanto mais horizontal for um *framework* mais genérico ele será, porém a aplicação poderá ser permeada com funcionalidades não necessárias àquela situação.

Uma outra forma de classificação dos *frameworks* é em relação às técnicas utilizadas para a sua especialização [23]. Seguindo esses critérios os *frameworks* são classificados em:

- *Frameworks* de caixa branca: nessa categoria a especialização do *framework*, para uso em uma situação particular do domínio, é realizada através de recursos da orientação a objetos tais como herança e ligação dinâmica. Os *hot-spots* são especializados através da herança de classes abstratas e sobreposição de métodos definidos pelo *framework*. Essa classe de *frameworks* requer que o desenvolvedor tenha conhecimento acerca da estrutura interna definida pelo *framework*.
- *Frameworks* de caixa preta: nessa categoria a especialização do *framework* é realizada através de interfaces que permitem a conexão de componentes à estrutura interna do *framework*. Os *hot-spots* são especializados através da implementação de componentes em conformidade com uma interface em particular e da integração destes componentes ao *framework*.
- *Frameworks* de caixa cinza: essa categoria contém *frameworks* que tentam aproveitar as vantagens

das abordagens da caixa branca e da caixa preta, ou seja, procuram prover soluções flexíveis e extensíveis, retendo ao mesmo tempo informações desnecessárias ao desenvolvedor.

2.3.2 ACE (ADAPTIVE Communication Environment)

O ACE (*ADAPTIVE Communication Environment*) [90] é um *framework* horizontal para desenvolvimento de sistemas distribuídos desenvolvido pelo *DOC Group* [37]. ADAPTIVE é o acrônimo de "*A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment*". O ACE disponibiliza uma série de soluções para os problemas mais comuns no desenvolvimento de sistemas distribuídos, tais como manipulação de sinais, inicialização de serviços, comunicação entre processos (*Inter-Process Communication - IPC*), gerenciamento de memória compartilhada e sincronização, dentre outros.

Dentre os benefícios de uso do ACE, destacam-se:

- Portabilidade: migrar uma solução baseada no ACE de uma plataforma de sistema operacional para outra é uma tarefa facilitada, devido ao uso de macros que adequam o código-fonte às nuances de um determinado sistema operacional. O ACE atualmente é utilizado com sucesso em uma série de sistemas operacionais, incluindo o Windows, Mac OS, Linux, VxWorks e QnX.
- Qualidade: o projeto do ACE foi baseado no uso de padrões de projeto que possibilitam a construção de sistemas flexíveis, extensíveis e modulares. Dentre os principais padrões de projeto definidos e implementados no ACE destacam-se aqueles utilizados para demultiplexação de eventos, inicialização de serviços e gerenciamento de concorrência [19].
- Eficiência e previsibilidade: o ACE foi projetado de modo a suportar diversos requisitos de qualidade de serviço (QoS), incluindo latência reduzida, desempenho em aplicações com tráfego intenso em rede e previsibilidade para aplicações de tempo-real.

A estrutura do ACE é dividida em camadas com objetivos específicos, ilustradas na figura 2.7.

No nível mais baixo, a camada de adaptação ao sistema operacional realiza o mapeamento das interfaces, específicas de um determinado sistema operacional, em interfaces padronizadas pelo ACE. Com essa abordagem, as camadas superiores ficam independentes de detalhes e mecanismos de um determinado sistema operacional, proporcionando uma maior portabilidade das aplicações. Dentre os mecanismos padronizados por essa camada destacam-se aqueles relacionados a concorrência, sincronização e sistemas de arquivos.

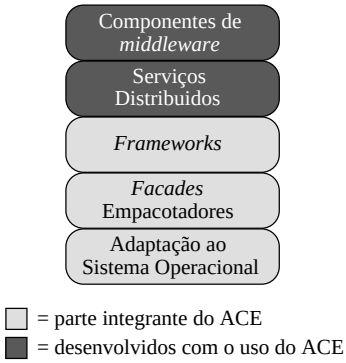


Figura 2.7: Integrantes e utilizadores do ACE

Na camada acima, os *facades* empacotadores têm o objetivo de disponibilizar uma biblioteca orientada a objetos que substitui a utilização das chamadas de sistema padronizadas pela camada de adaptação. Essa biblioteca encapsula as funções do sistema operacional, proporcionando abstrações que facilitam a utilização destes recursos. O ACE disponibiliza *facades* para operações tais como concorrência e sincronização de componentes (*mutexes*, semáforos etc), comunicação local e remota de processos (*sockets*, *pipes*, etc), sistemas de arquivos e gerenciamento de memória (compartilhada ou dinamicamente alocada). O ACE utiliza uma série de técnicas que minimizam o *overhead* introduzido por essas abstrações, tais como a utilização de declarações *inline*[4] (evitando o *overhead* de invocações de métodos) e a não utilização de métodos virtuais em *facades* críticos, tais como as operações de *send* e *receive* em *sockets* ou arquivos.

Na camada do *framework* são definidas arquiteturas que suportam a configuração dinâmica de aplicações distribuídas. As principais funcionalidades desse *framework* são: o tratamento de eventos gerados a partir de operações de I/O, *timers*, sinais ou operações de sincronização; e a configuração estática ou dinâmica de aplicações, provendo a substituição de módulos e facilidades para o desenvolvimento de aplicações baseadas em camadas, a exemplo de pilhas de protocolos em nível de aplicação.

Apesar de não fazerem parte da arquitetura do ACE, diversos serviços distribuídos, freqüentemente utilizados nas aplicações, estão disponíveis para operações, tais como localização de objetos, *logging*, roteamento de eventos e sincronização de relógios. Além de constituírem blocos básicos para o desenvolvimento de sistemas distribuídos, são exemplos práticos de uso dos padrões de projeto e *frameworks* disponibilizados pelo ACE.

Apesar dos benefícios obtidos com o uso do ACE, o desenvolvimento de aplicações distribuídas robustas, extensíveis e eficientes ainda é uma tarefa que envolve o conhecimento de diversos aspectos. Para auxiliar no gerenciamento dessa complexidade, a utilização de soluções de *middleware* tem sido uma prática rotineira. Como exemplos de soluções de *middleware* que utilizam o ACE pode-se citar o TAO (*The ACE ORB*) [91] e o *JAWS Adaptive Web Server* [47].

[4] Execução caracterizada pela substituição da invocação da função pela inserção direta do código-fonte, reduzindo o *overhead* gerado. Esta operação é realizada automaticamente pelos compiladores modernos.

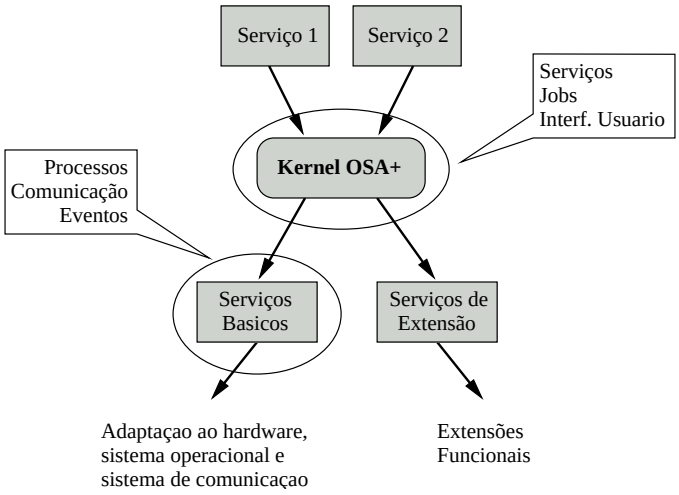


Figura 2.8: A arquitetura OSA+
Fonte: [8]

2.4 Abordagem baseada em objetos

A seção 2.2 deste trabalho apresentou as principais diferenças entre as tecnologias de objetos e componentes. Apesar das vantagens oferecidas pela tecnologia de componentes, soluções de *middleware* baseadas em objetos ainda são amplamente utilizadas no desenvolvimento de sistemas.

Diversos modelos de objetos e arquiteturas de *middleware* para sistemas de tempo-real foram propostos nos últimos anos. Esses trabalhos geralmente consistem ou na definição de novos modelos e arquiteturas voltados para tempo-real ou na adequação de arquiteturas já consolidadas, de modo a identificar e otimizar os pontos de imprevisibilidade dessas soluções. Dentre as soluções de *middleware* para tempo-real propostas, pode-se destacar o OSA+ (*Open System Architecture PLatform for Universal Services*) [8], o TMOSM (*Time-Triggered Message-Triggered Object Support Middleware*) [51, 52, 53], *ICa Middleware* [86] e o TAO (*The ACE ORB*) [91].

O OSA+ é um *middleware* projetado para o desenvolvimento de sistemas de tempo-real embarcados com facilidades para distribuição. A plataforma OSA+ é formada por um *kernel* bastante reduzido que oferece funcionalidades básicas e não contém partes dependentes do hardware ou do sistema operacional. Conforme ilustrado na figura 2.8, o OSA+ utiliza uma camada de serviços básicos que servem para realizar a adaptação do *kernel* a um ambiente específico de hardware e sistema operacional. Além dos serviços básicos, o *kernel* possibilita a definição de serviços da aplicação (indicados na figura como Serviço 1 e Serviço 2) e disponibiliza uma interface de usuário simples, utilizada para monitoramento dos serviços. Funcionalidades adicionais podem ser incluídas através da implementação de serviços de extensão.

Serviços de aplicação são criados no OSA+ através da função apresentada no código 2.4:

O parâmetro *type* define o modelo de execução utilizado e os valores possíveis são: *procedure* (função executada no mesmo espaço de endereçamento do processo inicial), *lightweightprocess* (função executada

Código 2.4: Exemplo de criação de tarefa de tempo-real no OSA+

```
1 osaStartService (void (* proc)(), osaProcType type,
2                   osaConfigType* config, char *name);
```

em uma nova *thread*) ou *process* (função executada em um novo processo). O valor de *config* ajusta parâmetros adicionais de configuração, tais como as restrições temporais. Se nenhuma condição temporal é especificada, todas as tarefas são tratadas com a mesma prioridade, usando uma abordagem FIFO. Os parâmetros temporais do serviço são representados pela estrutura apresentada no código 2.5.

Código 2.5: Especificação de requisitos temporais no OSA+

```
1  typedef struct {
2     struct {
3        osaTime deliver_at, deliver_tolerance, deliver_period;
4        osaLong deliver_mult;
5     } order_mode;
6     union {
7        struct {
8           osaTime deadline;
9           osaShort critically;
10       } realtime;
11       struct {
12          osaShort priority;
13          osaTime timeslice;
14       } realtime_priority;
15       struct {
16          osaShort cpu_load;
17       } realtime_load;
18    } service_mode;
19 } osaMode;
```

A estrutura *order_mode* define as restrições do canal de comunicação, ao passo que a estrutura *service_mode* define as restrições da execução do serviço. *deliver_at* e *deliver_tolerance* especificam o intervalo no qual a mensagem deve ser entregue. Se esse tempo é excedido, a mensagem é retornada para o emissor juntamente com uma mensagem de erro. O atributo *deliver_period* especifica o intervalo de tempo depois do qual a mensagem deve ser enviada novamente e *deadline* contém o último instante de tempo no qual a resposta da mensagem deve ser entregue. *critically* informa o que fazer se um *deadline* expira (*soft*, *firm* ou *hard*). *priority* é uma forma alternativa de especificação da restrição temporal através de prioridades. *cpu_load* contém a porcentagem de utilização da CPU requerida pelo serviço, permitindo que a execução do serviço seja independente da carga atual da CPU.

Uma outra abordagem de *middleware* para tempo-real é o *Time-Triggered Message-Triggered Object Support Middleware* (TMOSM), desenvolvido a partir de 1994. O TMO (*Time-Triggered Message-Triggered Object*) estende o modelo convencional de objetos, definindo uma estruturação sintática e semântica para a especificação dos requisitos temporais.

O TMOSM é a implementação que suporta o conceito do TMO. As primeiras implementações do TMOSM foram realizadas utilizando um sistema operacional de tempo-real denominado *DREAM Kernel*,

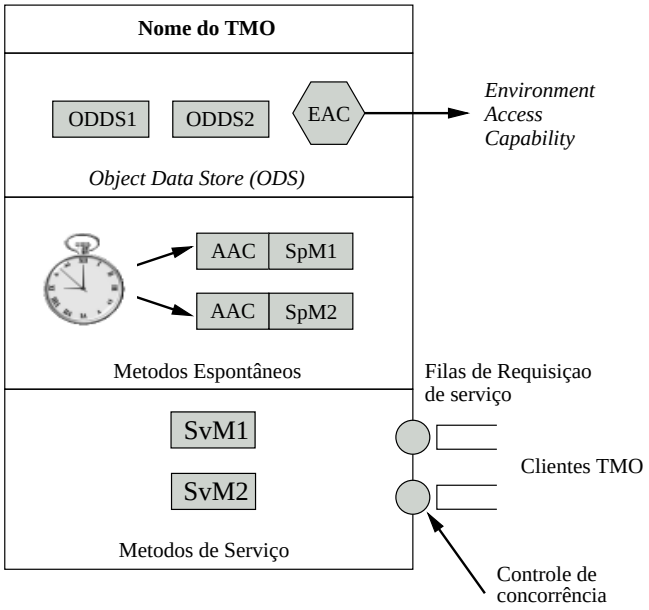


Figura 2.9: Estrutura básica de um TMO
Fonte: [51]

porém atualmente os autores têm direcionado os trabalhos para a construção de um *middleware* de tempo-real que funcione sobre plataformas COTS. Atualmente, existe uma versão disponível para a plataforma Windows NT.

As principais características desse *middleware* são: comunicação baseada em mensagens UDP de *broadcast*, concorrência e controle realizados através de técnicas não-bloqueantes e escalonamento de tarefas realizado em dois níveis. A figura 2.9 ilustra a estrutura básica de um TMO. As suas principais características são:

- Definição de atributos e recursos para comunicação: cada TMO possui uma área de dados onde atributos podem ser criados sob a forma de segmentos. O *Environment Access Capability* (EAC) oferece mecanismos para acessar o ambiente de comunicação em rede, dispositivos de I/O e outros TMO's.

- Definição de métodos espontâneos: o TMO contém um tipo de método chamado *time-triggered method* (TM) ou métodos espontâneos, em contrapartida aos métodos convencionais, aqui chamados de métodos de serviço. A execução de métodos espontâneos é disparada em tempos pré-definidos durante o projeto, ao passo que métodos de serviço são executados a partir de requisições dos clientes.

- Restrição básica de concorrência (*Basic Concurrency Constraint* - BCC): métodos de serviço não podem interferir na execução de métodos espontâneos. Basicamente, a ativação de um método de serviço solicitado por algum cliente é realizada somente quando não existem execuções potenciais

de métodos espontâneos.

- Definição e garantia de restrições temporais: para cada método espontâneo de um TMO é especificada uma janela de tempo na qual a execução deve iniciar e terminar, indicado na figura 2.9 com a sigla ACC, a ser descrita a seguir.

Os tempos de execução de métodos espontâneos devem ser especificados como constantes durante o projeto do sistema. Eles aparecem na primeira cláusula da especificação dos métodos e são chamados de condição de ativação autônoma (*Autonomous Activation Condition* - AAC). O código 2.6 apresenta a forma de definição de um método espontâneo no TMOSM.

Código 2.6: Especificação de requisitos temporais no TMOSM

```
1 for <time-var> = from <activation-time> to <deactivation-time>
2                                                    [ every <period> ]
3     start-during ( <earliest-start-time>, <latest-start-time>)
4                                                    finish-by <deadline>
```

Por exemplo, o código 2.7 especifica que esse método espontâneo deve ser executado a cada 150 ms, initerruptamente (das 0h às 23h59min). Cada execução deve iniciar em qualquer tempo dentro do intervalo (t, t+5ms) e deve ser completada até t+150 ms.

Código 2.7: Exemplo de especificação de requisitos temporais no TMOSM

```
1 for t = from 0am to 11:59pm every 150ms
2     start-during (t, t+5ms) finish-by t+150ms
```

2.4.1 TAO (The ACE ORB)

Muitas aplicações de tempo-real podem se beneficiar de arquiteturas flexíveis e abertas para computação distribuída, tais como aquelas definidas pela especificação CORBA. O TAO é uma implementação do CORBA 2.x, com extensões para tempo-real, desenvolvido a partir de 1997 pelo *DOC Group*.

Dentre os requisitos e características da implementação de ORB's necessários para aplicações de tempo-real, destacam-se: políticas e mecanismos para especificação de requisitos de qualidade de serviço, garantia dessa qualidade de serviço através de recursos do sistema operacional e do sub-sistema de comunicação, protocolos de comunicação eficientes e previsíveis, demultiplexação e *dispatching* eficientes e previsíveis, *stubs* e *skeletons* eficientes e previsíveis e gerenciamento de memória eficiente e previsível através da minimização de cópias de dados e alocações dinâmicas. Esses objetivos são trabalhados no TAO através de uma análise detalhada de cada componente da arquitetura CORBA.

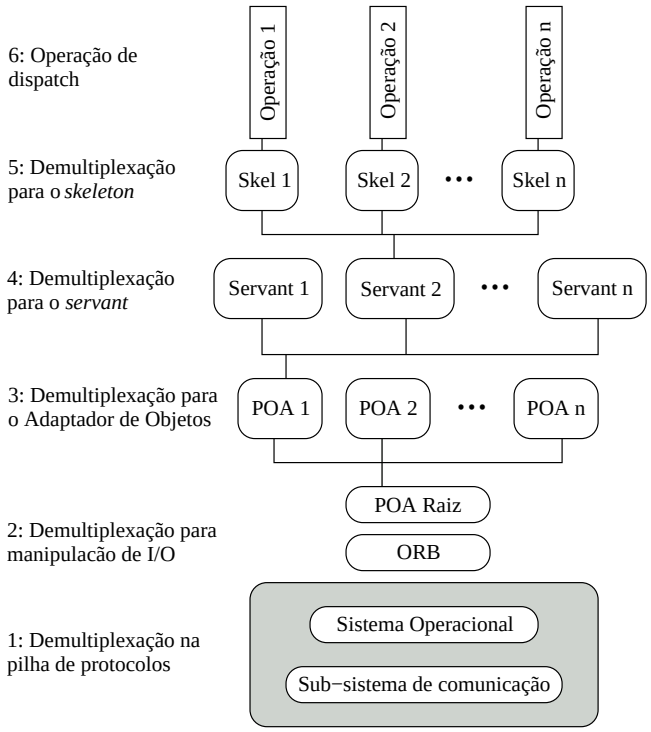


Figura 2.10: Abordagem tradicional do CORBA 2.x para demultiplexação de requisições
Fonte: [91]

A previsibilidade em relação ao ORB foi tratada através da substituição do protocolo IIOP (*Internet Inter-ORB Protocol*) por um protocolo que permite a especificação de parâmetros de qualidade de serviço. Esse protocolo é o RIOP (*Real-Time Inter-ORB Protocol*) e faz uso de campos já existentes e não utilizados do IIOP para enviar informações de controle da previsibilidade.

Na arquitetura CORBA 2.x, a alocação de requisições de clientes para *servants* pré-estabelecidos é realizada através de uma operação chamada demultiplexação. A demultiplexação geralmente é um ponto crítico de imprevisibilidade devido à sua freqüente abordagem hierárquica de resolução, conforme ilustrado na figura 2.10. Para localizar o *servant* e o método a ser executado, diversas operações de demultiplexação são realizadas em vários níveis. O TAO contorna esse problema através de duas técnicas alternativas de demultiplexação: *hashing* perfeito e demultiplexação ativa.

Na primeira abordagem, uma função de *hash* perfeito é utilizada para, automaticamente e em tempo linear, encontrar um *servant* disponível. Encontrado um *servant* disponível, uma segunda função de *hash* perfeito é utilizada para encontrar o método dentro do *servant*. O benefício dessa estratégia é que a busca de *servants* e métodos requer o tempo $O(1)$ [5] no pior caso. O TAO usa a ferramenta GNU *gperf* [89] para gerar as funções de *hash* perfeito.

Na demultiplexação ativa, o cliente informa um identificador de objeto que está diretamente associado ao *servant* e o método em questão. O cliente obtém esse identificador quando ele recebe uma referência de objeto através de um servidor de nomes, por exemplo. Em contraste ao *hashing* perfeito, a demulti-

[5] Notação matemática utilizada para analisar o comportamento assintótico de algoritmos.

plexação ativa não requer que todos os identificadores de objeto sejam conhecidos *a priori*. Isto a torna mais adequada para aplicações que definem objetos CORBA 2.x dinamicamente.

A obtenção de *stubs* e *skeletons* previsíveis se torna possível através da otimização em relação às cópias de dados e alocações dinâmicas e substituição de chamadas excessivas de funções por execuções *inline*.

O TAO, além de implementar os serviços comumente utilizados em sistemas distribuídos, tais como serviço de nomes, eventos, segurança e persistência, disponibiliza serviços fundamentais para o desenvolvimento de aplicações de tempo-real, a exemplo do Serviço de Escalonamento e o Serviço de Eventos de Tempo-Real.

2.4.1.1 Serviço de escalonamento

A especificação e garantia das restrições temporais no TAO é suportada através do *Scheduling Service* [33]. Esse serviço é responsável pela alocação de recursos de CPU de modo a satisfazer as necessidades temporais das aplicações. O *Scheduling Service* é definido como um objeto CORBA, ou seja, como uma implementação de uma interface IDL. Essa abordagem permite que o serviço seja acessado local ou remotamente sem implicações para os clientes.

O *Scheduling Service* tem duas principais atividades, uma realizada *off-line* e outra *on-line*:

- Análise *off-line* da escalonabilidade: o *Scheduling Service* realiza um teste *off-line* de escalonabilidade para toda operação registrada no serviço. Esse teste avalia se existem recursos de CPU suficientes para realizar todas as operações críticas.
- Consulta *on-line* de prioridades de requisições: em tempo de execução, o *Scheduling Service* disponibiliza uma interface que permite ao ORB a consulta de prioridades atribuídas previamente de forma *off-line*.

Para a especificação dos requisitos de qualidade de serviço, a linguagem de definição de interfaces original do CORBA foi estendida, dando origem à RIDL (*Real-Time Interface Definition Language*). Através dessa nova linguagem, programadores podem especificar os requisitos temporais de suas aplicações sob a forma de *deadlines*, periodicidade das tarefas, tempos de execução no pior caso etc. A especificação dos requisitos temporais é realizada através da utilização da interface *RT_ Task* e da estrutura *RT_Info*, ambas definidas pelo TAO, conforme indicadas no código 2.8.

Componentes que requerem garantias temporais podem especificar a QoS desejada herdando a interface *RT_ Task* e especificando os atributos da estrutura *RT_ Info*. O Serviço de Escalonamento atualmente trabalha somente com prioridades fixas e pode ser configurado para utilizar os algoritmos RMS

Código 2.8: Especificação de requisitos temporais no TAO

```
interface RT_Task {
   typedef Time::TimeT TimeT;
   typedef Time::TimeT PeriodT;
   enum Task_Priority { INTERRUPT, IO_SERVICE, CRITICAL, HARD_DEADLINE, BACKGROUND };
                        // From highest to lowest.
   struct RT_Info {
      TimeT worst_case_execution_time_;
      TimeT typical_execution_time_;
      PeriodT period_;
      Task_Priority priority;
      TimeT quantum;
      sequence < RT_Info > task_dependencies_;
   };
};
```

(*Rate-Monotonic Scheduling*) [80] ou MUF (*Maximum Urgency First*) [97] para escalonamento de tempo-real. Para uma discussão sobre as políticas de escalonamento adotadas pelo TAO e sobre a sua integração com sistemas operacionais de tempo-real vide seção 3.4.

2.4.1.2 Serviço de eventos de tempo-real

Grande parte das aplicações CORBA 2.x utiliza o modelo tradicional de comunicação cliente/servidor denominado *twoway* [59]. Nesse modelo, o cliente realiza a invocação do método e permanece bloqueado até que o servidor execute toda a operação e, eventualmente, envie para o cliente algum valor de retorno. Além disso, essa abordagem exige que o cliente gerencie as referências para o servidores com os quais ele deseja se comunicar. Em casos onde é necessário uma comunicação do tipo muitos-para-muitos, o gerenciamento dessas referências pode ser impraticável. Em sistemas de tempo-real, é comum a situação onde muitos sensores precisam se comunicar com um controlador ou este controlador precisa enviar sinais para diversos atuadores. O modelo de comunicação baseado em eventos proporciona uma forma desacoplada, flexível e robusta para promover essa comunicação do tipo muitos-para-muitos.

O OMG especificou um serviço CORBA 2.x denominado *COS*[6] *Event Service*. Nesse serviço, um mediador denominado Canal de Eventos é responsável pelo recebimento de mensagens criadas por objetos produtores e pelo repasse destas mensagens a objetos consumidores. Os produtores e consumidores mantêm somente a referência para o Canal de Eventos. Diversos produtores e consumidores podem ser conectados a um mesmo Canal de Eventos e este se responsabiliza pelo gerenciamento das referências, bem como o repasse das mensagens a todos os consumidores conectados àquele canal.

O *COS Event Service* define dois modelos para entrega de eventos: o modelo *push* e o modelo *pull* [59]. No modelo *push*, produtores enviam mensagens para o Canal de Eventos e este realiza a entrega destas mensagens aos consumidores conectados. No modelo *pull*, a requisição ocorre no sentido contrário: consumidores solicitam mensagens ao Canal de Eventos e este solicita mensagens aos produtores

[6] COS = *Common Object Services*.

conectados. Vale ressaltar que, no modelo *push*, consumidores devem ser implementados como objetos distribuídos, para que estes tenham seus métodos invocados pelo Canal de Eventos. Já no modelo *pull*, produtores devem ser implementados como objetos distribuídos, já que agora estes terão seus métodos invocados pelo Canal de Eventos.

Para a implementação de produtores e consumidores *push*, objetos CORBA 2.x devem ser criados através da implementação de interfaces disponibilizadas pelo *COS Event Service*. O código 2.9 apresenta as interfaces padronizadas pelo *COS Event Service* para a criação de produtores e consumidores *push*.

Código 2.9: Interfaces para implementação de produtores e consumidores do *COS Event Service*

```
1  module CosEventComm {
2     exception Disconnected {};
3
4     interface PushConsumer {
5        void push (in any data) raises (Disconnected);
6        void disconnect_push_consumer();
7     };
8
9     interface PushSupplier {
10       void disconnect_push_supplier();
11    };
12 };
```

A interface *PushConsumer* (linhas 4 a 7) define os métodos a serem implementados pelos consumidores: o *push()*, invocado pelo Canal de Eventos para entrega de uma nova mensagem e o *disconnect_push_consumer()*, invocado pelo Canal de Eventos para indicar que o consumidor em questão será imediatamente desconectado. A informação armazenada na mensagem sendo entregue é representada pelo argumento *data* do método *push()*. Devido ao fato desse argumento utilizar o tipo *any*, qualquer informação pode ser trafegada no Canal de Eventos, desde tipos primitivos até referências a outros objetos.

A interface *PushSupplier* (linhas 9 a 11) define o método a ser implementado por produtores: o *disconect_push_supplier()*, invocado pelo Canal de Eventos para indicar que o produtor em questão será imediatamente desconectado.

Conforme apresentado na figura 2.11, o Canal de Eventos disponibiliza interfaces *proxy* para realizar a comunicação entre produtores e consumidores. Com essa abordagem, produtores e consumidores de eventos se comportam como se estivessem se comunicando, respectivamente, com o consumidor e produtor envolvido no processo. Para a obtenção desses *proxies*, produtores e consumidores devem fazer uso de interfaces definidas pelo *COS Event Service*, apresentadas no código 2.10.

Após a inicialização do *COS Event Service*, o objeto que implementa a interface *EventChannel* deverá estar ativo e provavelmente registrado no Servidor de Nomes. A partir desse objeto, as interfaces *proxies* podem ser obtidas.

O código 2.11 apresenta o protocolo padrão para conexão de consumidores *push*. Após a criação

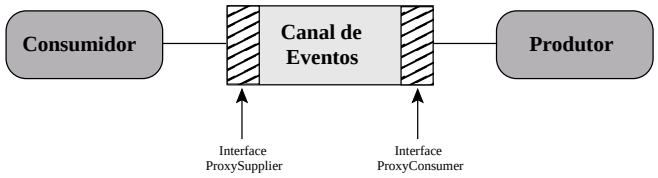


Figura 2.11: Interfaces *proxy* para comunicação de produtores e consumidores
Fonte: [59]

Código 2.10: Interfaces para obtenção dos *proxies* de produtores e consumidores *push*

```
1  module CosEventChannelAdmin {
2      interface ProxyPushSupplier;
3      interface ProxyPushConsumer;
4
5      interface ConsumerAdmin {
6          ProxyPushSupplier obtain_push_supplier();
7      };
8
9      interface SupplierAdmin {
10         ProxyPushConsumer obtain_push_consumer();
11     };
12
13     interface EventChannel {
14         ConsumerAdmin for_consumers();
15
16         SupplierAdmin for_suppliers();
17         void destroy();
18     };
19 };
```

(linha 3) e ativação (linha 4) do objeto que representa o consumidor, o objeto *ConsumerAdmin* é obtido através da invocação do método *for_consumers()*, disponibilizado pelo Canal de Eventos (linha 7). O próximo passo consiste na obtenção do *proxy* para o produtor[7], através da invocação do método *obtain_push_supplier()*, disponibilizado pelo objeto *ConsumerAdmin* (linhas 10 e 11). Após a obtenção do *proxy*, o consumidor é conectado ao Canal de Eventos através da invocação do método *connect_push_consumer()* (linha 14). Com a realização desses passos, o método *push()* do objeto consumidor conectado será devidamente invocado sempre que algum produtor enviar uma mensagem para o Canal de Eventos.

Código 2.11: Protocolo padrão para conexão de consumidores *push*

```
1  // Criando um novo objeto CORBA 2.x para representar o consumidor
2  // Este objeto implementa a interface PushConsumer
3  PushConsumer_impl servant(orb);
4  CosEventComm::PushConsumer_var my_consumer = servant._this();
5
6  // Assume-se que a variavel channel armazena a referencia para o Canal de Eventos
7  CosEventChannelAdmin::ConsumerAdmin_var consumer_admin = channel->for_consumers();
8
9  // Obtendo o proxy do produtor push a partir do ConsumerAdmin
10 CosEventChannelAdmin::ProxyPushSupplier_var supplier =
11                                                 consumer_admin->obtain_push_supplier();
12
13 // Realizando a conexao do consumidor, informando a referencia deste
14 supplier->connect_push_consumer(my_consumer);
```

[7] Lembre-se que consumidores trabalham com *proxies* para produtores e vice-versa.

O código 2.12 apresenta o protocolo padrão para envio de mensagens pelo produtores. Visto que, no modelo *push*, o Canal de Eventos invoca operações somente em consumidores, produtores geralmente não são implementados como objetos distribuídos nem necessitam de uma conexão ao Canal de Eventos. Eventualmente, um produtor pode ser implementado como um objeto e conectado no Canal quando existe a necessidade de notificação de desconexão, realizada pela invocação do método *disconnect_push_supplier()* do produtor em questão. Após a obtenção do *proxy* para o consumidor (linhas 2 a 6), uma mensagem contendo uma *string* é criada (linha 9) e enviada para o Canal de Eventos (linha 12). Após a recepção desta mensagem, o Canal de Eventos invoca o método *push()* em cada um dos consumidores devidamente conectados, passando como parâmetro a variável do tipo *CORBA::Any,* recebida do produtor.

Código 2.12: Protocolo padrão para envio de mensagens através de produtores *push*

```
1 // Assume-se que a variavel channel armazena a referencia para o Canal de Eventos
2 CosEventChannelAdmin::SupplierAdmin_var supplier_admin = channel->for_suppliers();
3
4 // Obtendo o proxy do consumidor push a partir do SupplierAdmin
5 CosEventChannelAdmin::ProxyPushConsumer_var consumer =
6                                                 supplier_admin->obtain_push_consumer();
7
8 // Criando a mensagem a ser enviada
9 CORBA::Any any << "Mensagem a ser enviada";
10
11 // Enviando a mensagem a todos os consumidores conectados no Canal de Eventos
12 consumer->push(any);
```

O *COS Event Service* representa uma boa alternativa para o desenvolvimento de sistemas distribuídos flexíveis, porém a sua utilização em sistemas de tempo-real tem sido impedida devido à ausência de mecanismos para especificação de requisitos temporais, bem como o *dispatch* de mensagens baseado em prioridades.

O Serviço de Eventos de Tempo-Real [42], disponibilizado pelo TAO, promove alterações e extensões no *COS Event Service* de modo a adaptá-lo para o uso em sistemas distribuídos de tempo-real. As principais melhorias realizadas são:

- Suporte à especificação de requisitos temporais: os produtores e consumidores informam ao Serviço de Eventos de Tempo-Real os seus requisitos temporais, tais como período, *deadline* e tempos de execução. Em associação com o Serviço de Eventos de Tempo-Real, o Serviço de Escalonamento executa os testes de escalonabilidade e atribui prioridades e sub-prioridades para cada participante conectado ao canal.

- Suporte ao *dispatch* de mensagens baseado em prioridades: em tempo de execução, o módulo de *dispatch* do Canal de Eventos de Tempo-Real consulta o Serviço de Escalonamento de modo a obter as prioridades dos participantes envolvidos. Essas prioridades são então utilizadas no processo de

entrega de mensagens.

- Mecanismos para filtragem e correlação de eventos: no *COS Event Service*, todos os consumidores conectados recebem mensagens de todos os produtores. No Serviço de Eventos de Tempo-Real, mensagens podem ser filtradas por tipo, produtor ou ambos. Além disso, mensagens podem ser correlacionadas, permitindo a criação de grupo de eventos baseados em conjunções e disjunções.

- Suporte a eventos de *timeout*: o Canal de Eventos de Tempo-Real pode ser configurado para, espontaneamente, gerar eventos de um determinado tipo, os quais podem ser entregues a consumidores interessados nesta informação. O canal de eventos se comporta, desta forma, como um produtor autônomo, gerando eventos numa freqüência determinada pelo desenvolvedor.

De modo a possibilitar a entrega de eventos baseada em prioridades, produtores e consumidores devem informar, ao Serviço de Escalonamento, as restrições temporais exigidas na sua execução. Conforme apresentado na sub-seção 2.4.1.1, essas restrições são armazenadas em estruturas do tipo *RT_Info*. O código 2.13 apresenta a conexão de um consumidor *push* no Serviço de Eventos de Tempo-Real do TAO.

Código 2.13: Protocolo para conexão de consumidores *push* no Serviço de Eventos de Tempo-Real

```
1  // Criando um novo objeto CORBA 2.x para representar o consumidor
2  // Este objeto implementa a interface RtecEventComm::PushConsumer
3  RtecPushConsumer_impl servant(orb);
4  RtecEventComm::PushConsumer_var my_consumer = servant._this();
5
6  // Assume-se que a variavel channel armazena a referencia para o Canal de Eventos
7  RtecEventChannelAdmin::ConsumerAdmin_var consumer_admin = channel->for_consumers();
8
9  // Obtendo o proxy do produtor push a partir do ConsumerAdmin
10 RtecEventChannelAdmin::ProxyPushSupplier_var supplier =
11                                                 consumer_admin->obtain_push_supplier();
12
13 // Criando uma nova estrutura RT_Info. Assume-se que a variavel scheduler armazena
14 // a referencia para o Servico de Escalonamento
15 RtecScheduler::handle_t consumer_rt_info = scheduler->create ("consumer");
16
17 // Ajustando as restricoes temporais do consumidor
18 scheduler->set (
19     consumer_rt_info,                    // Nome da estrutura RT_Info
20     RtecScheduler::VERY_LOW_CRITICALITY, // Criticalidade
21     tmp,                                 // Tempo no pior caso (Worst-Case Execution Time)
22     tmp,                                 // Tempo tipico (medio)
23     tmp,                                 // Tempo com uso de cache
24     time_val_to_period (tv),             // Periodo
25     RtecScheduler::VERY_LOW_IMPORTANCE,  // Importancia
26     tmp,                                 // Quantum (nao utilizado)
27     0,                                   // No de threads
28     RtecScheduler::OPERATION             // Tipo da estrutura RT_Info
29 );
30
31 // Informando para o objeto ConsumerQOS a estrutura RT_Info a ser utilizada na conexao
32 ACE_ConsumerQOS_Factory consumer_qos;
33 consumer_qos.insert_type (ACE_ES_EVENT_UNDEFINED, consumer_rt_info);
34
35 // Realizando a conexao do consumidor, informando
36 // a referencia deste e a estrutura RT_Info
37 supplier->connect_push_consumer(my_consumer, consumer_qos.get_ConsumerQOS());
```

Após a criação (linha 3) e ativação (linha 4) do objeto que representa o consumidor e a obtenção do *proxy* para o produtor (linhas 7 a 11), uma nova estrutura *RT_Info* é criada através da invocação do método *create()* do Serviço de Escalonamento (linha 15). Essa estrutura irá armazenar os parâmetros temporais do consumidor sendo conectado. Esses parâmetros são informados ao Serviço de Escalonamento através da invocação do método *set()* (linhas 18 a 29).

A estrutura *RT_Info* armazena informações importantes para a realização dos testes de escalonabilidade e do escalonamento propriamente dito, porém apenas uma parte destas informações são utilizadas por uma determinada estratégia de escalonamento. Por exemplo, ao configurar o Serviço de Escalonamento para uso do algoritmo RMS (*Rate Monotonic Scheduling*) apenas o parâmetro "período" é utilizado, enquanto o algoritmo MUF (*Maximum Urgency First*) utiliza apenas o parâmetro "criticalidade". Os parâmetros "tempo no pior caso", "tempo típico" e "tempo com uso de *cache*" são utilizados pelo Serviço de Escalonamento para a realização dos testes de escalonabilidade. Parâmetros tais como "importância", "*quantum*" e "número de *threads*" estão disponíveis caso outras estratégias de escalonamento sejam implementadas.

Após o ajuste dos parâmetros temporais, um objeto do tipo *ACE_ConsumerQOS_Factory* é criado para armazenar a estrutura *RT_Info* (linha 32). O consumidor é conectado no Serviço de Eventos de Tempo-Real através da invocação do método *connect_push_consumer()* (linha 36), que agora recebe, além da referência para o objeto que representa o consumidor, o objeto *ACE_ConsumerQOS_Factory*, que contém a estrutura *RT_Info*. Após a conexão de todos os produtores e consumidores do serviço, o método *compute_scheduling()* do Serviço de Escalonamento deve ser invocado, de modo a executar os testes de escalonabilidade e determinar as prioridades de entrega de mensagens aos consumidores.

Os módulos internos do Canal de Eventos de Tempo-Real estão ilustrados na figura 2.12. Após as operações de filtragem e correlação de eventos realizados pelos respectivos módulos, uma consulta é realizada ao *Scheduling Service* de modo a obter a prioridade e a sub-prioridade do evento em questão. O núcleo de *dispatch* é composto por filas com diferentes prioridades. A prioridade do evento indica em qual fila este evento será armazenado, enquanto a sub-prioridade indica a posição do evento dentro de uma determinada fila.

Todas as operações de armazenamento dos requisitos temporais de produtores e consumidores, cálculo do escalonamento das mensagens, bem como a entrega de prioridades e sub-prioridades de eventos é realizada por um *framework* denominado Kokyu, a ser descrito na sub-seção 2.4.1.3.

2.4.1.3 O framework Kokyu

O Kokyu[8] [14] é um *framework* portável que provê serviços flexíveis para escalonamento e *dispatch,*

[8] Palavra japonesa que literalmente significa "respiração", mas também indica algo coordenado, com execução ajustada.

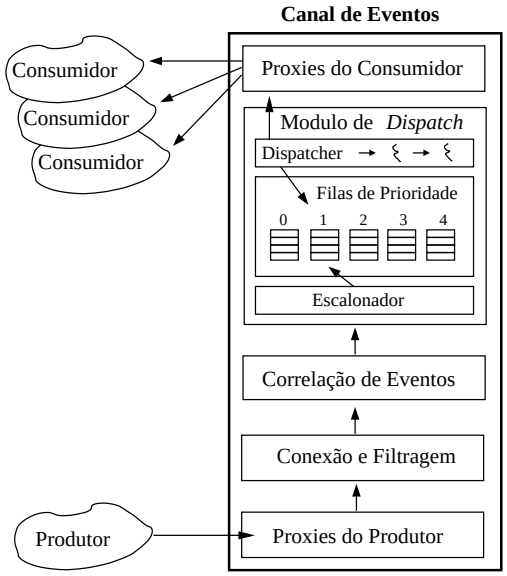


Figura 2.12: Estrutura interna do Serviço de Eventos de Tempo-Real do TAO
Fonte: [42]

e foi projetado de modo a ser utilizado em conjunto com soluções de *middleware*, tais como o TAO. O Serviço de Escalonamento disponibilizado pelo TAO, e que atua conjuntamente com o Serviço de Eventos de Tempo-Real, faz uso do Kokyu nas atividades de escalonamento e entrega de prioridades e sub-prioridades.

O *framework* Kokyu é formado por dois componentes principais:

- Uma estrutura modular para escalonamento que possibilita a utilização de diferentes estratégias, tais como escalonamento estático, dinâmico e híbrido.
- Uma estrutura flexível para *dispatch* que permite a composição de mecanismos do sistema operacional com serviços do *middleware*, de modo a garantir as estratégias de escalonamento.

O escalonador é responsável pela especificação de como as requisições são ordenadas, através da atribuição de prioridades. O *dispatcher* garante a execução ordenada dessas requisições, através da criação de filas e *threads* baseadas em prioridades.

Todos os requisitos temporais informados pela aplicação para o *framework* Kokyu são armazenadas na estrutura *RT_Info*, apresentada no código 2.8. O maior benefício proporcionado pelo *framework* Kokyu é a possibilidade de utilização de novas estratégias de escalonamento com um impacto mínimo para a aplicação.

2.5 Abordagem baseada em componentes

Nos últimos anos, diversos trabalhos têm sido realizados no intuito de avaliar e adequar a tecnologia de componentes de software para uso em sistemas de tempo-real [12]. Deseja-se manter os benefícios obtidos com a abordagem de componentes e, ao mesmo tempo, dotar o *middleware* da previsibilidade necessária à execução dos sistemas de tempo-real. Dentre os principais projetos nessa área, pode-se destacar o RCCF (*Real-Time Component Customization Framework*) [106, 107], o ACCORD (*AspeCtual COmponent based Real-time system Development*) [2], o Cadena [3, 48], o modelo e implementação propostos por Wang et. al. [104] e o CIAO (*Component-Integrated ACE ORB*) [103, 65].

O RCCF define um modelo para componentes de tempo-real, bem como um conjunto de ferramentas para configuração e implantação. O *framework* define componentes para as atividades principais de um sistema de tempo-real e novas funcionalidades podem ser implementadas através da especialização destes componentes. Apesar de oferecer uma boa flexibilidade de configuração em tempo de projeto, os mecanismos utilizados para a especialização tornam a abordagem restrita, devido ao fato de serem baseados em um conjunto fixo de propriedades configuráveis.

O ACCORD é um esforço no sentido de aplicar as teorias da orientação a aspectos no desenvolvimento de sistemas de tempo-real. O objetivo é modelar os requisitos temporais como aspectos que permeiam todo o sistema. As funcionalidades de componentes, bem como os aspectos, são implementados a partir de um conjunto pré-definido de primitivas, cujo valor de execução no pior caso é conhecido. Essa proposta, apesar de ter uma boa fundamentação teórica, não prevê a sua utilização em ambientes distribuídos.

O Cadena é um ambiente para desenvolvimento de sistemas baseados no CCM, formado por um conjunto de ferramentas para análise do comportamento temporal. O Cadena estende uma implementação em andamento do CCM, o OpenCCM [1], adicionando mecanismos que permitem a especificação dos requisitos temporais. O componente projetado e os seus requisitos temporais são então transformados e enviados para uma ferramenta de *model checking*, denominada *DSpin* [17], responsável pela verificação da viabilidade do escalonamento.

Wang et. al. desenvolveram um sistema de tempo-real baseado em componentes que integra a solução de *middleware* com os mecanismos básicos de sistemas operacionais e redes de tempo-real. Além disso, o trabalho também define políticas para controle de admissão de novas tarefas, permitindo modificações *on-line* no conjunto de tarefas. O sistema foi implementado utilizando o sistema operacional de tempo-real *TimeSys* [100] e o JBoss [38], uma implementação livre da especificação *Java 2 Plataform, Enterprise Edition* (J2EE), que contém o padrão EJB para componentes distribuídos.

O CIAO é uma implementação da especificação CCM, voltada para uso em sistemas distribuídos de tempo-real e foi a tecnologia adotada na implementação do *framework* aqui proposto. Essa escolha é justificada pelos seguintes pontos: utilização de tecnologias subjacentes já consolidadas na área de tempo-

real (ACE e TAO), implementação de funcionalidades CCM (não disponibilizadas ainda pelos outros trabalhos) importantes para a flexibilidade e reutilização da plataforma (configuração e implantação facilitada, modificação *on-line* da montagem, etc), possibilidade de uso em sistemas embarcados e suporte à heterogeneidade requerida pelas aplicações industriais. Nas sub-seções seguintes são apresentados, com mais detalhes, os mecanismos e serviços implementados no CIAO.

2.5.1 CIAO (Component-Integrated ACE ORB)

O CIAO (*Component-Integrated ACE ORB*) é uma implementação do CCM, com extensões para uso em sistemas de tempo-real. A implementação do CIAO é baseada na utilização do ACE e do TAO como suporte básico para a obtenção de uma solução previsível, portável e interoperável. Um dos objetivos do CIAO é prover o suporte a execução e comunicação previsível de componentes através da disponibilização de serviços não-funcionais, configuráveis através de descritores de implantação (arquivos XML).

O CIAO estende as especificações do CCM através da implementação de um servidor de componentes de tempo-real. Esse servidor é formado por um *container* que disponibiliza interfaces para o gerenciamento das políticas de QoS e para a interação com os mecanismos que garantem a execução previsível desses componentes.

O CIAO, assim como as outras implementações do CCM, são trabalhos atualmente em andamento e representam implementações parciais da especificação CCM. Além de implementar o modelo de componentes CCM descrito anteriormente, o CIAO disponibiliza uma solução para configuração e implantação de sistemas CCM (o DAnCE) e uma solução para modificações *on-line* na montagem (o ReDaC), apresentados a seguir. Essas tecnologias constituem pontos importantes para a manutenibilidade, flexibilidade e reutilização de sistemas baseados no CCM.

2.5.1.1 DAnCE (Deployment and Configuration Engine)

No desenvolvimento baseado em componentes, uma montagem pode ser formada pela implantação de diversos componentes em diferentes nós de uma rede. Essas montagens são organizadas em pacotes que contêm os componentes compilados e os arquivos descritores de implantação.

Durante o processo de implantação e execução de sistemas baseados em componentes, diversas atividades se fazem necessárias: implantação de componentes em nós remotos, ativação e desativação de componentes, inicialização e configuração de recursos do *container* e configuração dos serviços não-funcionais utilizados. De modo a padronizar essas operações, o *DOC Group* criou o DAnCE (*Deployment and Configuration Engine*) [31], uma implementação da especificação, criada pelo OMG, para implantação e configuração de componentes [72].

As funcionalidade básicas do DAnCE para suportar a implantação e configuração de sistemas baseados em componentes são:

- Representação em memória dos descritores de implantação: de modo a evitar *overheads* constantes durante as operações de configuração, o DAnCE realiza a leitura dos descritores de implantação uma única vez, durante o processo de implantação. Além disso, é mantida uma representação em memória do Plano de Implantação, facilitando acessos futuros.
- *Download* automático de pacotes de componentes: montagens podem ser modificadas dinamicamente através da migração de pacotes de um nó para outro da rede, mesmo em ambientes heterogêneos.
- Configuração automática de ORB's, *containers* e servidores de componentes: essas configurações são realizadas com o objetivo de atingir os requisitos de QoS especificados, reduzindo a possibilidade de erros caso essa configuração seja realizada pelo desenvolvedor.
- Conexão automáticas de *ports* de componentes: outras implementações do CCM delegam para o desenvolvedor a responsabilidade de conectar, através de código, os componentes da montagem. Com o CIAO, essas conexões são definidas no descritor de implantação e realizadas automaticamente pelo DAnCE.
- Implementação e gerenciamento automático de serviços não funcionais: serviços tais como eventos, segurança e balanceamento de carga são configurados pelo DAnCE, deixando o desenvolvedor livre para a correta implementação dos serviços funcionais da aplicação.

O DAnCE é formado por uma série de participantes, implementados como objetos CORBA 2.x convencionais, devido ao fato de o DAnCE não poder utilizar a sua própria arquitetura para implantar e configurar este serviço. Os principais participantes do DAnCE são:

- *ExecutionManager*: responsável pelo gerenciamento do processo de implantação em um ou mais domínios de implantação. Segundo a especificação para implantação e configuração do OMG, um domínio de implantação é definido como o conjunto de nós, componentes e conexões que compõem uma determinada montagem.
- *DomainApplicationManager*: responsável pelo gerenciamento do processo de implantação em um domínio particular. Esse participante divide o Plano de Implantação (descritor) em diversos subplanos, um para cada nó do domínio. No DAnCE, o *ExecutionManager* e o *DomainApplicationManager* são implementados no mesmo processo de modo a se beneficiar dos recursos do TAO para comunicação local de objetos.
- *NodeManager*: esse objeto é executado em cada nó que compõe o domínio e é responsável pelo gerenciamento das implantações designadas para aquele nó. Componentes CCM são criados por

containers que, por sua vez, são hospedados em processos servidores de componentes denominados *NodeApplications*. O *NodeManager* cria um objeto denominado *NodeApplicationManager*, o que é capaz de criar objetos *NodeApplications*.

- *NodeApplicationManager*: é implementado no mesmo processo do *NodeManager* e é utilizado para diferenciar implantações num mesmo nó do domínio. Para cada implantação um *NodeApplicationManager* é criado e é responsável pela recepção dos descritores daquela implantação.

- *NodeApplication*: executa o papel do servidor de componentes, no sentido de prover os recursos computacionais (CPU, memória e facilidades de comunicação) para a execução do componente. Esse objeto cria os *containers* que disponibilizam um ambiente para a instanciação dos componentes.

- *RepositoryManager*: é um objeto único para um determinado domínio de implantação, e é utilizado por desenvolvedores, para armazenar implementações de componentes, e pelo *NodeApplicationManager*, para obter, sob demanda, os componentes necessários para a implantação. Cada *NodeApplicationManager* utiliza o *RepositoryManager* para consultar implementações de componentes e copiá-los localmente no nó onde ele será implantado.

- *PlanLauncher*: programa executável responsável pela leitura do plano descritor de implantação e pelo envio deste plano ao *ExecutionManager*.

A figura 2.13 apresenta o procedimento de implantação e configuração, realizado pelo DAnCE, e os objetos envolvidos neste processo. No passo 1, o desenvolvedor solicita ao *ExecutionManager* o início do processo de implantação. Para cada nó definido no domínio de implantação, o *ExecutionManager* envia, para o *NodeApplicationManager,* a parte do descritor XML referente àquele nó e solicita a implantação dos componentes (passo 2). Os componentes em questão são adquiridos do *RepositoryManager* (passo 3) e os servidores de componentes são criados (passo 4). Após a configuração dos recursos do servidor de componentes (passo 5) e inicialização dos serviços de *middleware* requisitados (passo 6), o *container* é criado (passo 7) e os componentes e *homes* são efetivamente instalados (passo 8). O último passo (9) consiste na configuração dos atributos e conexões presentes nos componentes implantados.

Com a arquitetura para implantação e configuração definida pelo DAnCE, o processo de implantação e configuração de componentes em diversos nós de um domínio é substancialmente facilitado. O código 2.14 apresenta os passos necessários para implantar uma montagem no DAnCE. Os comandos abaixo são executados em uma sessão do terminal do sistema operacional.

As linhas 1 e 2 inicializam o Servidor de Nomes. A implementação do Servidor de Nomes do TAO contempla a utilização de mensagens de *multicast* para descoberta automática deste serviço. Se inicializado com este suporte (parâmetro '-m 1') o Servidor de Nomes não precisa ter o seu endereço explicitamente

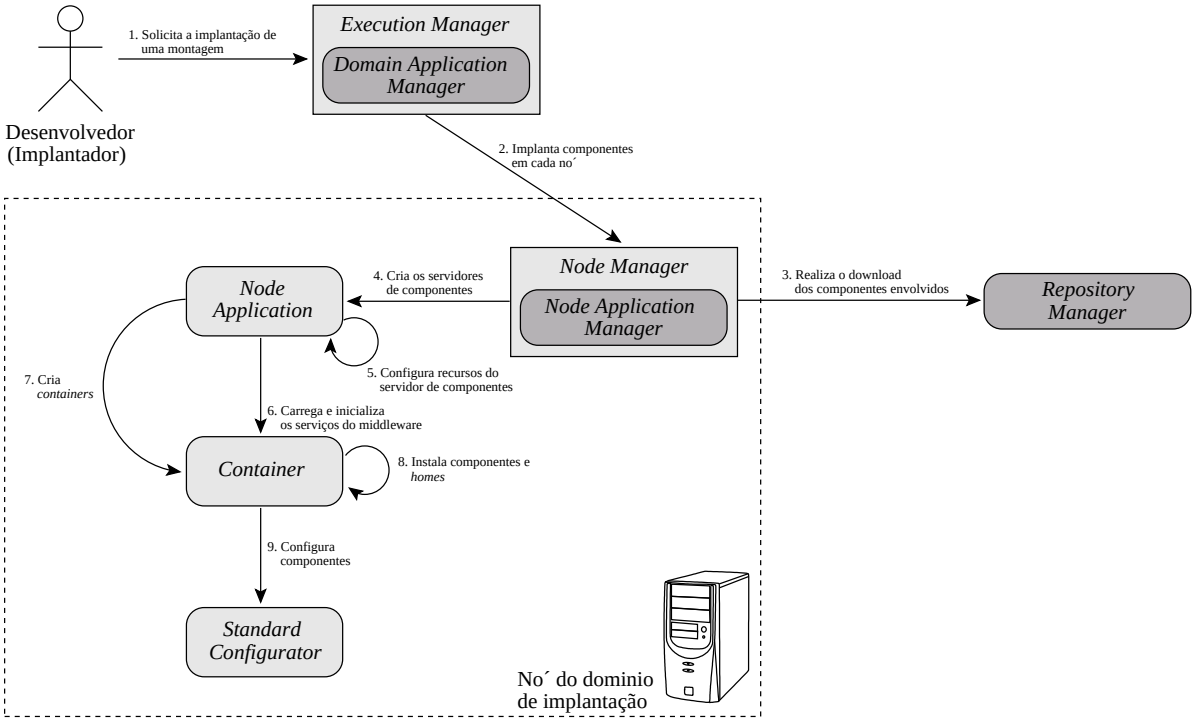


Figura 2.13: Objetos do DAnCE envolvidos no processo de implantação e configuração
Fonte: [31]

Código 2.14: Passos para implantação de uma montagem no DAnCE

```
1 > $TAO_ROOT/orbsvcs/Naming_Service/Naming_Service -m 0
2                                                   -ORBEndpoint iiop://localhost:60003 -o ns.ior
3 > $DAnCE/NodeManager/NodeManager -ORBEndpoint iiop://localhost:60001
4                                  -s $DAnCE/NodeApplication/NodeApplication -o NodeApp1.ior -d 30
5 > ...
6
7 > $DAnCE/ExecutionManager/Execution_Manager -o EM.ior -i NodeManagerMap.dat
8 > $DAnCE/Plan_Launcher/plan_launcher -p flattened_deploymentplan.cdp
9                                                                   -k file://EM.ior -o DAM.ior
10 > $DAnCE/Plan_Launcher/plan_launcher -k file://EM.ior -i file://DAM.ior
```

indicado em clientes. Para aplicações de tempo-real, entretanto, sugere-se a indicação explícita da localização do serviço, por motivos de *overhead* do uso do *multicast*. Desta forma, o Serviço de Nomes está sendo inicializado no exemplo sem suporte a *multicast* (parâmetro '-m 0'). O parâmetro *-ORBEndpoint* informa qual porta deverá ser utilizada por este serviço (60002), enquanto o parâmetro '-o' indica o nome do arquivo no qual será gravada a referência (*Interoperable Object Reference - IOR*) do servidor.

O próximo passo consiste em executar, em cada nó do domínio de implantação, o objeto *NodeManager*. As linhas 3 e 4 demonstram a inicialização desse objeto em um dos nós. O parâmetro *-ORBEndpoint* indica a porta a ser utilizada pelo objeto, enquanto o parâmetro '-s' indica o local do executável do *NodeApplication*. O parâmetro '-o' indica o nome do arquivo no qual será gravada a referência (IOR) desse *NodeManager* e o parâmetro '-d' indica o tempo máximo para aguardar a inicialização de cada *NodeApplication* criado pelo *NodeManager* (neste caso, 30 segundos).

A linha 7 apresenta a inicialização do *ExecutionManager*. O parâmetro '-o' indica o nome do arquivo

no qual será gravada a referência (IOR) desse objeto, enquanto o parâmetro '-i' indica qual o arquivo que contém o mapa de nós do domínio de implantação. Esse arquivo texto contém uma lista onde, para cada nó, são informados o seu nome e o endereço do *NodeManager* correspondente. O código 2.15 exemplifica a construção deste mapa de nós[9].

Código 2.15: Exemplo de mapa de nós para um determinado domínio de implantação

```
1 LocalNode     corbaloc:iiop:localhost:60001/NodeManager
2 RemoteNode    corbaloc:iiop:10.2.1.2:60002/NodeManager
```

As linhas 8 e 9 do código 2.14 realizam a implantação propriamente dita, através do *PlanLauncher*. O parâmetro '-p' informa qual o descritor de implantação a ser utilizado, o parâmetro '-k' informa a referência para o *ExecutionManager* e o parâmetro '-o' informa o nome do arquivo no qual será gravada a referência para o objeto *DomainApplicationManager*, criado pelo *PlanLauncher*.

Para realizar a desimplantação (*undeploy*) e destruir todos os objetos com sucesso, conforme apresentado na linha 10, utiliza-se o *PlanLaucher* com o parâmetro '-i', indicando a referência para o *DomainApplicationManager* criado.

Após esses passos, o DAnCE realiza todas as operações de instanciação, configuração e conexão de componentes.

O DAnCE, em conjunto com o CIAO, disponibiliza um ambiente robusto e produtivo para o desenvolvimento, configuração e implantação de aplicações distribuídas baseadas em componentes.

2.5.1.2 ReDaC (Redeployment and Reconfiguration)

A grande flexibilidade introduzida pelo uso de componentes seria sub-utilizada se não existisse a possibilidade da reconfiguração, em tempo de execução, de instâncias, atributos de instâncias e conexões, com o objetivo de otimizar o sistema ou substituir implementações por outras mais eficientes ou livres de erros.

O ReDaC (*Redeployment and Reconfiguration*) é um serviço disponibilizado pelo DAnCE para reimplantação de montagens e reconfiguração de componentes já implantados, sem a necessidade da interrupção da aplicação. Com o ReDaC é possível:

- instalar ou remover instâncias criadas a partir de tipos já conhecidos pela implantação;
- instalar instâncias criadas a partir de tipos ainda não declarados na montagem;

[9] O nome indicado neste mapa, para cada nó do domínio, é o mesmo nome a ser utilizado no arquivo descritor de implantação, na *tag node*, do bloco *instance*. Desta forma, é possível implantar a montagem de forma distribuída, em diversos nós do domínio de implantação.

- instalar ou remover instâncias em/de um *container* que já existe;
- instalar novas instâncias em um *container* dinamicamente criado;
- migrar componentes de um *container* para um *container* já existente ou dinamicamente criado;
- instalar novas instâncias em outro servidor de componentes no mesmo nó ou em um nó remoto;
- remover, adicionar ou substituir conexão de facetas/receptáculos ou produtores/depósitos de eventos.

O ReDaC pode ser utilizado de duas formas: através de execuções manuais do *PlanLauncher* ou via programação, através de métodos disponibilizados pelo *ExecutionManager*.

Na abordagem manual, o *PlanLaucher* pode ser utilizado para reimplantar e reconfigurar uma montagem que já está em execução. Para isso, é necessário que o Plano de Implantação seja manualmente editado de modo a incluir/excluir/modificar instâncias, atributos e conexões. Para indicar que deseja-se reimplantar e reconfigurar uma montagem já em execução, utiliza-se o parâmetro '-r' do *PlanLaucher*, informando o nome do arquivo descritor da nova implantação. O código 2.16 ilustra essa utilização.

Código 2.16: Exemplo de reimplantação através do *PlanLauncher*

```
1 $DAnCE/Plan_Launcher/plan_launcher -r modified_flattened_deploymentplan.cdp
```

O DAnCE, no momento da implantação, cria uma representação em memória do Plano de Implantação indicado no *PlanLauncher*. Essa representação é realizada através de um objeto do tipo *DeploymentPlan*, que contém métodos para manipulação de todas as *tags* presentes no descritor de implantação. O código 2.17 apresenta um exemplo de uso do ReDaC, via programação. O objeto do tipo *DeploymentPlan* é obtido invocando-se o método *getPlan()* do *ExecutionManager* e passando como parâmetro o UUID[10] do Plano de Implantação a ser adquirido (linhas 1 e 2). Esse plano de execução pode ser modificado, via código, para a incluir/excluir/modificar instâncias, atributos e conexões (Linhas 3 e 4). Após a modificação, pode-se solicitar a reimplantação através da execução do método *performRedeployment()* do *ExecutionManager*, o qual recebe como parâmetro o Plano de Implantação modificado (linha 5).

Código 2.17: Exemplo de reimplantação via código

```
1 ::Deployment::DeploymentPlan_var deployment_plan =
2                     this->execution_manager_->getPlan("ARCOSDAISServer_Plan_UUID_0001");
3 ::ARCOS::ReDaCUtils::add_instance(deployment_plan, ns_group_home_name.c_str(),
4                 "MainNode", "ARCOS-DAISDAGroupHome-mdd", ns_group_home_name.c_str());
5 this->execution_manager_->perform_redeployment(deployment_plan);
```

[10] Este UUID deve coincidir com o valor da *tag* UUID do bloco *Deployment:deploymentPlan* do arquivo descritor de implantação.

A possibilidade de reconfiguração e reimplantação de componentes de forma *on-the-fly* é uma característica importante e que promove uma flexibilidade muito maior à tecnologia de componentes. O ReDaC pode ser utilizado, por exemplo, para corrigir componentes defeituosos através de um mecanismo automático para detecção de falhas, reparo e reintegração de componentes. Conexões podem ser automaticamente manipuladas, de modo a adaptar o sistema à condições do ambiente de execução, consistindo um suporte básico para o desenvolvimento de arquiteturas adaptativas.

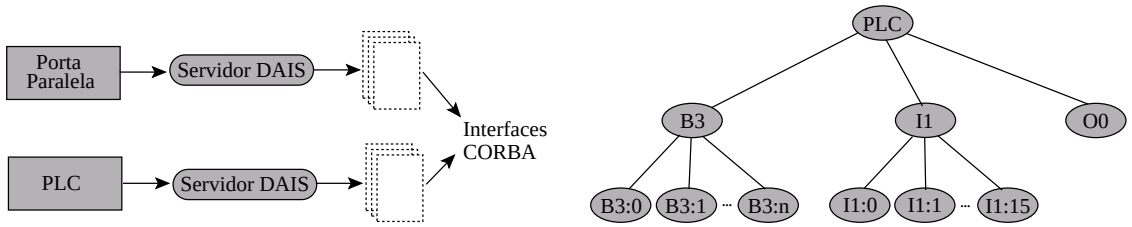
2.6 DAIS (Data Acquisition from Industrial Systems)

A definicão de padrões para interoperabilidade é uma preocupação crescente à medida em que as tecnologias de comunicação se tornam disponíveis e a heterogeneidade dos ambientes computacionais se torna inevitável. Na indústria não é diferente. A existência de soluções diversas, advindas de fabricantes distintos, representa um desafio para a total integração.

Uma padronização criada pelo OMG para a aquisição de dados industriais é o DAIS (*Data Acquisition from Industrial Systems*) [69]. O DAIS é uma especificação CORBA que permite a transferência de uma grande quantidade de dados industriais simultaneamente para vários clientes (consumidores). Esses dados industriais são identificados e incorporados com informações de caducidade (validade temporal). Dentre as principais funcionalidades disponibilizadas pela especificação pode-se ressaltar: a consulta de quais dados são disponibilizados pelo servidor DAIS, leitura e escrita de dados de forma síncrona e assíncrona, criação e manutenção de assinaturas (*subscriptions*) e uso de *iterators* e interfaces de *callback* para transferência de grandes volumes de dados. Apesar dessa especificação ser rotulada como uma padronização para **aquisição** de dados, estes podem também ser alterados e enviados de volta à planta, através das interfaces de escrita definidas pelo DAIS.

A figura 2.14a ilustra o principal objetivo do DAIS: mapear dados, obtidos de um dispositivo de aquisição específico, em interfaces CORBA padronizadas. Esse mapeamento disponibiliza os dados sob a forma de uma árvore (figura 2.14b), que pode ser percorrida e ter seus nós selecionados por clientes DAIS, para posterior aquisição. A especificação DAIS define três formas distintas para leitura de dados e duas formas distintas para escrita de dados, indicados na tabela 2.3. Os objetos e métodos indicados nesta tabela serão explicados a seguir.

Os principais objetos definidos pelo DAIS para a consulta, aquisição e alteração de dados são: o *DAISServer*, o *DAISDataAccessSession*, o *DAISNodeHome*, o *DAISNodeIterator*, o *DAISGroupHome*, o *DAISGroupManager* e o *DAISGroupEntryIterator*. O papel desempenhado, o momento de criação e as relações de dependência entre estes objetos são explicados a seguir.



(a) Mapeamento de um dispositivo específico em interfaces CORBA

(b) Exemplo de construção da árvore DAIS para aquisição de dados de um CLP

Figura 2.14: Objetivos do DAIS

Leitura de Dados	Escrita de Dados
Síncrona (bloqueante) através do método *sync_read()* do objeto *DAISGroupManager*. Os valores dos dispositivos da planta são retornados diretamente pelo método *sync_read()*.	Síncrona (bloqueante) através do método *sync_write()* do objeto *DAISGroupManager*. Esse método termina sua execução quando todos os dados são atualizados em todos os dispositivos envolvidos.
Assíncrona (não-bloqueante) através do método *async_read()* do objeto *DAISGroupManager*. Os valores dos dispositivos da planta são retornados através do método *on_read_complete()* do objeto *Callback*, definido pelo cliente.	Assíncrona (não-bloqueante) através do método *async_write()* do objeto *DAISGroupManager*. Esse método termina sua execução quando a solicitação de escrita chega ao servidor DAIS. Após a atualização dos dados em todos os dispositivos envolvidos, uma invocação é realizada ao método *on_write_complete()* do objeto *Callback* do cliente, indicando a efetiva atualização.
Assíncrona (não-bloqueante) através de assinaturas (*subscriptions*). Com esse mecanismo, os dados são periodicamente enviados pelo servidor através de invocações ao método *on_data_change()*, do objeto *Callback* disponibilizado pelo cliente.	

Tabela 2.3: Formas distintas para leitura e escrita de dados no DAIS

O *DAISServer* é um *singleton*[11] [22] que está disponível durante todo o tempo em que o serviço está ativo e representa o ponto inicial de contato com clientes DAIS. A partir desse objeto, o cliente DAIS pode realizar as demais operações e adquirir referências para os outros objetos. Ao implementar um servidor DAIS o desenvolvedor deve organizar os dados a serem expostos sob a forma de uma árvore, e esta é armazenada na estrutura interna do objeto *DAISServer*. O uso de uma árvore para armazenamento dos dados expostos é justificado pelo fato de constituir uma estrutura bastante flexível, adequada para a representação de dados em um vasto conjunto de sistemas industriais. Essa árvore pode indicar a disponibilidade não só de dados do ambiente (temperatura, pressão, nível, etc), mas também informações adicionais tais como o modelo do sensor ou sua localização geográfica.

Antes de iniciar qualquer operação no servidor DAIS o cliente precisa solicitar a criação de um objeto que gerencia todo o ciclo de uso do servidor por aquele cliente em particular. Esse objeto se chama

[11] Padrão de projeto utilizado para restringir a instanciação de uma classe a somente um objeto.

DAISDataAccessSession e é criado através da invocação do método *create_data_access_session()* do objeto *DAISServer*. Esse objeto de sessão é mantido ativo até que o cliente encerre a sessão DAIS, invocando o método *destroy()* do objeto *DAISDataAccessSession*.

O objeto *DAISNodeHome* provê os métodos necessários para a construção e visualização da árvore DAIS de um servidor. Esse objeto é um *singleton* que existe durante toda a execução do serviço DAIS e é obtido através da invocação do método *node_home()* do objeto *DAISDataAccessSession*. O *DAISNodeHome* disponibiliza métodos para a obtenção do nó raiz da árvore DAIS (*get_root()*) e para a obtenção de todos os filhos de um determinado nó (*find_by_parent()*). Com esses dois métodos, uma implementação recursiva permite que um cliente DAIS visualize toda a árvore de dados disponibilizada pelo servidor. Além disso, métodos para a construção da árvore DAIS, tais como o *add_node()*, são também definidos pelo *DAISNodeHome*.

Com o objetivo de evitar invocações remotas demoradas, potenciais causadoras de imprevisibilidades temporais, o resultado do método *find_by_parent()* é um objeto do tipo *DAISNodeIterator*. Esse objeto permite a recuperação gradativa e controlada dos filhos de um determinado nó da árvore DAIS. A cada invocação do método *next_n()* do *DAISNodeIterator*, no máximo os n (parâmetro do método) próximos filhos do nó em questão são retornados, numa abordagem comumente conhecida como *Lazy Load*. Esses *iterators* não apresentam métodos para retrocesso e existem no servidor até que o seu método *destroy()* seja invocado.

No DAIS, a unidade de aquisição de dados é chamada de **grupo**. Ao criar um grupo, o cliente DAIS informa quais nós da árvore DAIS pertencem a este grupo, além de informar parâmetros temporais, como a freqüência de aquisição. A criação de grupos de aquisição é realizada pelo objeto *DAISGroupHome*, obtido através da invocação do método *get_home()* do objeto *DAISDataAccessSession*. Cada sessão (cliente) presente no servidor DAIS está associada a uma instância particular do *DAISGroupHome*, possibilitando que cada cliente especifique seus grupos particulares de aquisição. O objeto *DAISGroupHome* existe enquanto existir o objeto *DAISDataAccessSession* associado. Além dos grupo particulares criados por cada cliente, a especificação também prevê a criação de grupo públicos de dados, comuns a vários clientes de um mesmo servidor DAIS.

Novos grupos de dados são criados através da invocação do método *create_data_access_group()* do objeto *DAISGroupHome*. Para cada grupo criado, o *DAISGroupHome* instancia um objeto do tipo *DAISGroupManager* e retorna esta referência para o cliente. O objeto *DAISGroupManager* concentra todas as operações de aquisição e alteração dos dados industriais. Esse objeto permite a inclusão de folhas DAIS naquele grupo em particular (método *create_entries()*), indicando quais dados deverão ser adquiridos. A entrega de dados DAIS é realizada através de invocações periódicas a métodos de objetos de *callback*, fornecidos pelo cliente[12]. Através do método *cllbck()* do objeto *DAISGroupManager*,

[12] Quando utilizando a forma de entrega de dados via assinaturas (*subscriptions*).

o cliente é capaz de passar para o servidor a referência para este objeto de *callback*. A escrita de dados nos dispositivos é realizada através dos métodos *sync_read()* e *async_read()*, disponibilizados pelo *DAISGroupManager*. Um grupo de aquisição, e conseqüentemente o objeto *DAISGroupManager* associado, existe até que o método *destroy()* do *DAISGroupManager* seja executado.

Para um determinado grupo, é possível consultar as folhas DAIS já inseridas naquele grupo através da invocação do método *create_group_entry_iterator()*. Numa abordagem similar os *iterators* de nós da árvore DAIS, esse método retorna um objeto do tipo *DAISGroupEntryIterator*, que possibilita a consulta gradativa e controlada das folhas DAIS participantes do grupo. Esses *iterators* não apresentam métodos para retrocesso e existem no servidor até que o seu método *destroy()* seja invocado.

A figura 2.15 resume o protocolo utilizado no DAIS para a criação de grupos de aquisição de dados. Após todas essas operações, o método *on_data change()* do objeto de *callback* é invocado na freqüência informada durante a criação do grupo, atualizando o cliente sobre as mudanças dos dados desejados.

A especificação DAIS é dividida em três grandes grupos de funcionalidades: *DAIS Server*, *DAIS Data Access* e *DAIS Alarms and Events*. O *DAIS Server* especifica as operações necessárias para a criação de sessões, grupos de dados e as demais operações de comunicação com o cliente. O *DAIS Data Access* provê os mecanismos necessários para a entrega periódica de dados, definição da taxa de aquisição e especifica a interface de *callback* utilizada pelo cliente para a recepção de dados. O *DAIS Alarms and Events* especifica interfaces para a definição e recepção de dados advindos de alarmes e outros eventos do sistema industrial. Segundo a especificação, uma implementação DAIS para ser considerada em conformidade com o padrão:

- deve implementar o *DAIS Server*;
- deve implementar o *DAIS Data Access* OU o *DAIS Alarms and Events*;
- pode implementar o *DAIS Data Access* E o *DAIS Alarms and Events*.

Outros padrões para aquisição de dados foram propostos nos últimos tempos. Dentre eles, destaca-se o OPC (*OLE for Process Control*) [28]. Apesar da grande aceitação na indústria, o OPC apresenta como desvantagem a sua limitação ao uso em plataformas proprietárias, restringindo o escopo de interoperabilidade desejado. O OPC é uma padronização que influenciou sobremaneira o projeto do DAIS, visando uma futura integração entre as especificações.

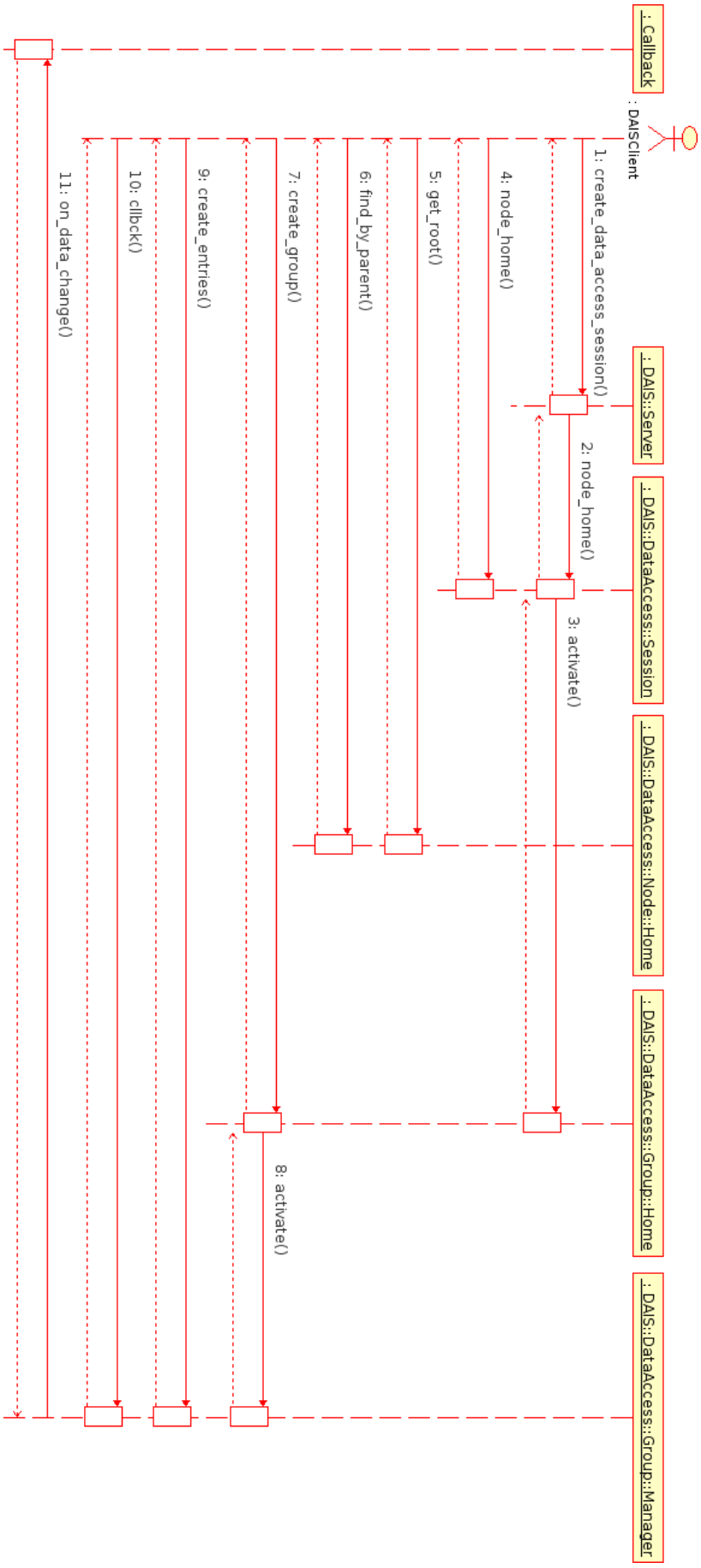


Figura 2.15: Diagrama de seqüência do processo de aquisição de dados no DAIS

Capítulo 3

ARCOS - ARquitetura para COntrole e Supervisão

Apesar de todos os benefícios trazidos com a adoção dos micro-controladores e CLP's, o modelo de programação utilizado por estas soluções se limita a programas cíclicos e mono-programados, desenvolvidos em linguagens de difícil manutenção, tais como LADDER, *Assembly* ou C. A utilização de paradigmas já consolidados da Ciência da Computação, como a orientação a objetos, ainda é um desafio. O alto custo das soluções baseadas em CLP's, aliado à necessidade de um ambiente computacional mais flexível e que possibilite soluções mais modernas, tem recentemente justificado o desenvolvimento de pesquisas interessadas na utilização de componentes COTS de hardware e de software, diminuindo os custos e favorecendo questões de interoperabilidade e de manutenção.

3.1 Objetivos do ARCOS

Visto sob este prisma, a evolução e modernização dos sistemas industriais de tempo-real é um processo que depende da disponibilização de ambientes de hardware, software e comunicação que sejam poderosos, flexíveis, de fácil manutenção e interoperáveis, ao mesmo tempo em que mantenham as características já presentes nos sistemas industriais atuais tais como robustez, tolerância a falhas e previsibilidade. Inserida nesse processo, a Ciência da Computação, e em particular a engenharia de software, estuda mecanismos e metodologias que possibilitam a construção dos ambientes de software a serem utilizados nos sistemas industriais modernos. Por outro lado, a crescente complexidade inserida no processo de desenvolvimento e no papel desempenhado pelo software demanda a prospecção de tecnologias e ferramentas que auxiliem o gerenciamento desta complexidade.

O ARCOS (ARquitetura para COntrole e Supervisão), plataforma de software proposta e implementada neste trabalho, é um *framework* (vide seção 2.3) para sistemas de S&C que facilita o desenvolvimento de sistemas industriais modernos. A seguir serão apresentados os principais desafios do desenvolvimento

de sistemas industriais modernos e quais foram as decisões tomadas, no projeto do ARCOS, para superá-los parcial ou totalmente.

Desafio 1: adoção de um modelo de programação flexível e poderoso. A demanda por sistemas industriais distribuídos, inteligentes e adaptativos tem sido o grande fator que motiva a adoção de uma nova infra-estrutura de software e hardware na indústria. A utilização de micro-controladores, CLP's e linguagens tais como LADDER e *Assembly* é uma entrave para a construção de sistemas com o grau de complexidade gerado pelos requisitos atuais. A orientação a objetos e as tecnologias de componentes e de *middleware* representam alternativas já consolidadas em áreas tais como os Sistemas de Informação, constituem soluções apropriadas para o atendimentos dos requisitos atuais da indústria [30, 43, 44, 108, 109] e foram as tecnologias de base para o projeto do ARCOS.

Desafio 2: suporte à distribuição e à interoperabilidade. A utilização das redes de comunicação de dados na indústria foi motivada por três fatores principais: inerente dispersão geográfica das manufaturas, redução dos custos de manutenção ainda causada pelo grande número de fios ligando sensores e atuadores aos CLP's, e necessidade de integração do chão-de-fábrica com os níveis superiores da manufatura (controle distribuído e sistemas de planejamento e gerenciamento da produção). A utilização de padrões abertos para comunicação industrial favorece a integração de soluções advindas de diferentes fornecedores. Entretanto, para a real integração da manufatura faz-se necessário a utilização de padrões abertos em todas as camadas dos sistemas, desde a comunicação de dados até os protocolos utilizados no nível de aplicação. Conforme apresentado na sub-seção 2.1.1, o CORBA é a solução para computação distribuída que provê maior suporte à heterogeneidade, no sentido em que possibilita a comunicação entre objetos independente da linguagem de programação, sistema operacional e arquiteturas de hardware e comunicação utilizados. Devido à heterogeneidade inerente e crescente dos ambientes industriais, o CORBA foi adotado como o *middleware* de suporte à distribuição utilizado no projeto do ARCOS. Além disso, o CORBA vem sendo cada vez mais utilizado como solução de integração em sistemas industriais [95, 88].

Desafio 3: melhoria da produtividade e da manutenção de sistemas industriais. O CORBA 2.x sempre foi conhecido como uma tecnologia de difícil utilização, de interesse exclusivo de instituições de pesquisa e por muitas vezes considerada decadente. Apesar da existência de soluções reais na indústria baseadas em CORBA 2.x [88], as limitações apresentadas na sub-seção 2.1.1 contribuíram para a disseminação deste preconceito. Com o lançamento da especificação CORBA 3 e do CCM, essas limitações foram eliminadas, através da definição do *container* executor de componentes (o desenvolvedor não necessita mais implementar todo o servidor, somente o componente em questão) e do uso de arquivos XML, ao invés de código-fonte, para indicar a utilização de serviços não-funcionais. O CCM é um modelo de componentes que, em conjunto com uma boa implementação e boas ferramentas, constitui um ambiente produtivo e de fácil manutenção para a construção de sistemas distribuídos modernos. O CCM,

em conjunto com a potencial reutilização proporcionada pela abordagem dos *frameworks*, caracterizam o ARCOS como um ambiente produtivo para o desenvolvimento de sistemas industriais modernos.

Desafio 4: utilização de um mecanismo flexível para computação distribuída. O alto acoplamento das unidades de implementação, em sistemas complexos, é um dos principais fatores que dificultam a manutenção e reutilização destas soluções. A implementação de unidades auto-contidas e com papéis bem definidos, em conjunto com um mecanismo flexível para definição de relacionamentos entre essas unidades é a principal abordagem utilizada pela tecnologia de componentes. A possibilidade de indicação desses relacionamentos (conexões) através de descritores XML poupa o desenvolvedor de codificações explícitas e sujeitas a erros. Além disso, um novo modelo de manutenção e de reutilização é instalado neste ambiente de desenvolvimento. Componentes de software defeituosos podem ser substituídos ou reparados e reintegrados. Demandas atuais da indústria, tais como as Células Flexíveis de Manufatura (*FMC - Flexible Manufacturing Cells*) [35], requerem a possibilidade de alteração facilitada dos processos, constituindo um bom alvo para aplicação da tecnologia de componentes, e justificando a escolha pelo CCM no projeto do ARCOS.

Desafio 5: utilização de um mecanismo flexível para comunicação distribuída. Na indústria é comum a situação onde um sensor precisa enviar uma informação para diversos sistemas, tais como um controlador, um supervisório ou um sistema de armazenamento histórico. A necessidade de um modelo de comunicação do tipo muitos-para-muitos, desacoplado, não-bloqueante e com flexibilidade de inclusão e retirada de produtores e consumidores motivou a utilização do Serviço de Eventos no projeto do ARCOS. Esse modelo flexível de comunicação, em conjunto com o modelo flexível de computação proporcionado pelas conexões de componentes CCM, possibilita a reutilização do ARCOS em um conjunto de aplicações industriais de S&C.

Desafio 6: suporte à interoperabilidade na aquisição de dados. A existência de diversas tecnologias de aquisição de dados, tais como CLP's e placas de aquisição através da porta paralela, sempre foi um fator agravador da efetiva integração e interoperabilidade de plantas industriais. O desenvolvimento de sensores inteligentes, passíveis de serem conectados diretamente em uma rede industrial, amenizou esse problema no sentido em que eles estão em conformidade com padrões de comunicação em rede. Entretanto, para a integração plena, é necessário a utilização de padronizações nos protocolos utilizados pelas **aplicações** que manipulam tais dados. O OMG define especificações baseadas no CORBA, dedicadas à padronização de interfaces voltadas para domínios de aplicação específicos, tais como telefonia e controle de tráfego aéreo. O DAIS (*Data Acquisition from Industrial Systems*) [69] é uma especificação CORBA que padroniza as interfaces utilizadas para consulta e aquisição de dados industriais. Ao disponibilizar uma implementação reutilizável de um servidor DAIS, o ARCOS realiza a disponibilização de dados, coletados da planta, para qualquer cliente em conformidade com o padrão DAIS. Apesar de ser um padrão com poucas implementações atuais, o DAIS apresenta todas as funcionalidades necessárias

à manipulação de dados industriais e apresenta possibilidades de integração com outros padrões, mais consolidados porém não baseados no CORBA, como por exemplo o OPC (*OLE for Process Control*) [28].

Desafio 7: especificação das restrições temporais e garantia da previsibilidade. O conservadorismo da indústria, em relação à adoção de soluções modernas de software, é justificado pela imprevisibilidade e baixa robustez apresentadas pelo sistemas computacionais convencionais (sistemas de informação). O desenvolvimento de ambientes previsíveis e tolerantes a falhas é fator crucial para a aplicação dessas novas soluções no ambiente industrial. Diversos estudos recentes, na área de Tempo-Real, propõem teorias e tecnologias para levar a previsibilidade de execução e de comunicação aos ambientes computacionais multi-programados. O ACE (*ADAPTIVE Communication Environment*) [90], o TAO (*The ACE ORB*) [91] e o CIAO (*Component-Integrated ACE ORB*) [65, 103] são exemplos de tecnologias ativas e com reais aplicações no desenvolvimento de sistemas de tempo-real [88]. A utilização do CIAO na implementação do ARCOS é justificada pelo fato de ser uma solução baseada em tecnologias já consolidadas no desenvolvimento de sistemas de tempo-real, além de constituir atualmente a implementação mais completa do CCM, se comparada com outras implementações tais como o MICO-CCM [79] e OpenCCM [1].

Desafio 8: configuração e implantação facilitada. Além dos esforços dispensados no projeto e implementação de sistemas distribuídos, a configuração e implantação destes sistemas, bem como o gerenciamento do seu correto funcionamento é uma atividade que apresenta alguns desafios. A implantação e realocação de componentes em diversos nós da rede envolve atividades como configuração de ORB's, ativação e conexão de componentes e configuração dos serviços utilizados. No CIAO, essas atividades são realizadas pelo DAnCE (*Deployment And Configuration Engine*) [31], apresentado na sub-seção 2.5.1.1. O DAnCE realiza a configuração e implantação remota de componentes, bem como todas as configurações necessárias para conexões e uso de serviços não-funcionais, poupando o desenvolvedor de tal atividade. Como deseja-se que aplicações baseadas no ARCOS se beneficiem dessas características, o DAnCE foi o mecanismo de configuração e implantação adotado neste projeto.

Desafio 9: suporte à adaptação dinâmica. Os sistemas adaptativos [75] são caracterizados pela capacidade de alterar a sua estrutura e funcionamento internos, em função de eventos ocorridos no ambiente operacional destes sistemas. Essa adaptação é importante em sistemas que tentam otimizar o seu funcionamento através de utilização de diversas estratégias para um mesmo objetivo. Como exemplos, pode-se citar a adaptação de um sistema embarcado com o objetivo de reduzir o consumo de energia ou a adaptação de um sistema de controle de modo a melhorar a qualidade do produto sendo gerado. Para prover adaptação, o sistema computacional deve utilizar uma arquitetura de software flexível e que permita essas modificações, até mesmo sem intervenção humana e sem interrupções no funcionamento da aplicação. O ReDaC, apresentado na sub-seção 2.5.1.2, é uma API disponibiliza pelo CIAO, que possibilita a reconfiguração e reimplantação de sistemas CCM já em funcionamento, sem interrupção do

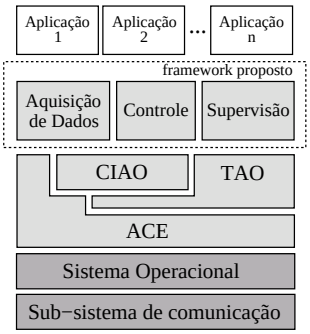


Figura 3.1: O ARCOS e suas tecnologias subjacentes

serviço. Com o ReDaC é possível substituir componentes, redefinir conexões e reconfigurar atributos de montagens já implantadas, constituindo um bom arcabouço para uma futura implementação do suporte à adaptação no ARCOS. Além disso, o ReDaC foi o mecanismo utilizado extensivamente no ARCOS para a criação de novas instâncias de componentes, operação a ser detalhada na sub-seção 3.3.3.

Desafio 10: disponibilização de ferramentas amigáveis para o desenvolvedor e para o usuário final da aplicação. Além de projetar e implementar toda a arquitetura *server-side* para uso em sistemas industriais, o ARCOS disponibiliza ainda ferramentas gráficas de auxílio ao uso do ARCOS e de utilização dos serviços providos pelo *framework*. Essas ferramentas permitem a visualização gráfica de uma planta industrial, papel reconhecidamente desempenhado pelos sistemas SCADA (*Supervisory Control And Data Acquisition*), além de auxiliar o processo de uso do ARCOS para o desenvolvimento de novos sistemas de S&C.

A figura 3.1 apresenta as tecnologias subjacentes utilizadas pela plataforma ARCOS. Nas camadas inferiores estão presentes o sub-sistema de comunicação em tempo-real e o sistema operacional de tempo-real, provendo os mecanismos básicos que garantem a previsibilidade do sistema. Na camada intermediária, encontram-se as tecnologias de implementação do ARCOS (ACE, TAO e CIAO), responsáveis pelo ambiente de execução de componentes distribuídos. Fazendo uso dessas três tecnologias, o ARCOS define serviços reutilizáveis para as atividades de aquisição de dados, controle e supervisão. Finalmente, na camada superior encontram-se as aplicações industriais que reutilizam a solução arquitetural definida pela plataforma ARCOS.

Nas seções seguintes serão apresentados, de forma detalhada, o modelo arquitetural proposto pelo framework ARCOS, questões técnicas da implementação realizada e algumas considerações sobre a relação do ARCOS com aspectos teóricos fundamentais para o desenvolvimento de sistemas de tempo-real distribuídos.

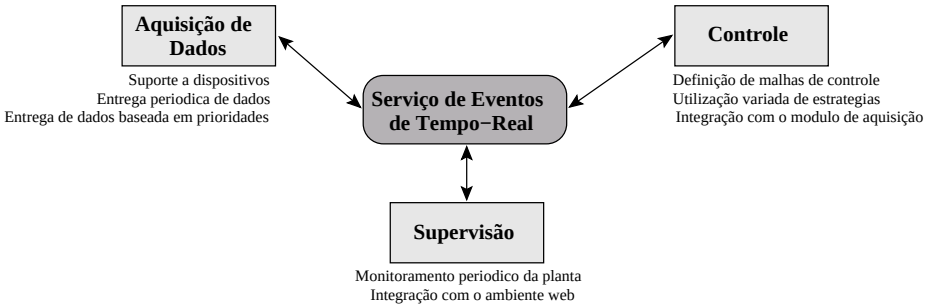


Figura 3.2: O Serviço de Eventos de Tempo-Real como elemento integrador dos componentes de aquisição de dados, controle e supervisão do ARCOS

3.2 O modelo

Nesta seção serão apresentados os componentes definidos para as atividades de aquisição de dados, controle e supervisão, os pontos de inversão de controle presentes no ARCOS e os *hot-spots* a serem especializados de modo a direcionar o ARCOS para uma situação específica de aquisição de dados e de controle.

Na arquitetura projetada para o ARCOS, o Serviço de Eventos de Tempo-Real desempenha um papel integrador da comunicação entre os diversos componentes propostos, possibilitando uma comunicação flexível e a entrega de mensagens baseada em prioridades. A figura 3.2 ilustra esta configuração. Com essa abordagem, além de integrar as soluções apresentadas na figura, é possível a inclusão futura de novos módulos no ARCOS, através da conexão de novos produtores e consumidores de eventos. Neste momento, é importante definir, de um modo geral, os papéis (produtores e consumidores) desempenhados pelos módulos de aquisição de dados, controle e supervisão.

O módulo de aquisição de dados, representado pela implementação da especificação DAIS, em grande parte do tempo executa um papel de produtor de eventos, enviando periodicamente para clientes DAIS os dados coletados da planta. Entretanto, ao receber atualizações de dados representando atuações na planta, esse módulo se comporta como um consumidor de eventos.

O módulo de controle é um cliente DAIS que realiza o papel de consumidor e produtor de eventos. Ele consome mensagens produzidas pelo módulo de aquisição de dados, executa os devidos cálculos de controle, e produz informações de atuação que serão entregues, via Serviço de Eventos, de volta ao módulo de aquisição.

O módulo de supervisão é, essencialmente, um consumidor de mensagens no sentido em que permite o monitoramento da situação do chão-de-fábrica. Entretanto, eventos podem ser produzidos, por exemplo, para alterar o valor do *setpoint* de uma malha de controle.

Nas seções seguintes serão apresentados os componentes participantes de cada um destes módulos, bem como os mecanismos que permitem a utilização do ARCOS em uma variedade de situações de aquisição de dados e controle.

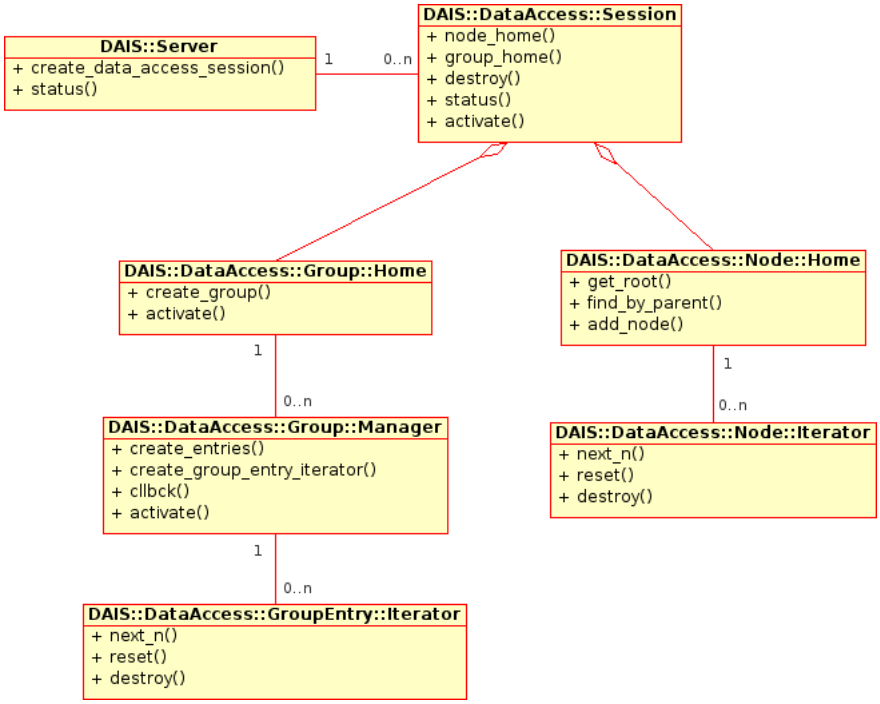


Figura 3.3: Diagrama simplificado dos objetos definidos pela especificação DAIS

3.2.1 Aquisição de dados

O módulo de aquisição de dados do ARCOS é composto essencialmente pela implementação da especificação DAIS e pela definição do *hot-spot* de aquisição de dados, a serem explicados a seguir. Devido ao fato de o DAIS ser uma especificação CORBA 2.x, uma reestruturação, baseada em componentes, desta especificação foi projetada e implementada neste projeto, sem entretanto gerar impactos na implementação de clientes DAIS. Esses clientes se comunicarão com o servidor como se fosse implementado a partir de objetos CORBA 2.x convencionais. Essa característica é importante para questões de interoperabilidade com clientes DAIS já existentes. Essa implementação baseada em componentes permite a reutilização desse serviço em situações diversas, cada uma delas caracterizada por uma tecnologia distinta para aquisição de dados (CLP's, porta paralela, redes industriais, etc).

A implementação DAIS realizada no ARCOS contempla os grupos de funcionalidades *DAIS Server* e *DAIS Data Access*, constituindo uma solução em conformidade com a especificação. Dentre as formas distintas de leitura e escrita de dados, são contempladas, no modelo proposto pelo ARCOS, a leitura de dados via assinaturas e a escrita bloqueante e não-bloqueante de dados, via métodos *sync_read()* e *async_read()*, respectivamente.

Conforme apresentados na seção 2.6, os principais objetos definidos pelo DAIS para a consulta e aquisição de dados são: o *DAISServer*, o *DAISDataAccessSession*, o *DAISNodeHome*, o *DAISNodeIterator*, o *DAISGroupHome*, o *DAISGroupManager* e o *DAISGroupEntryIterator*. A figura 3.3 apresenta um diagrama de classes simplificado, resumindo os relacionamentos entre estes objetos.

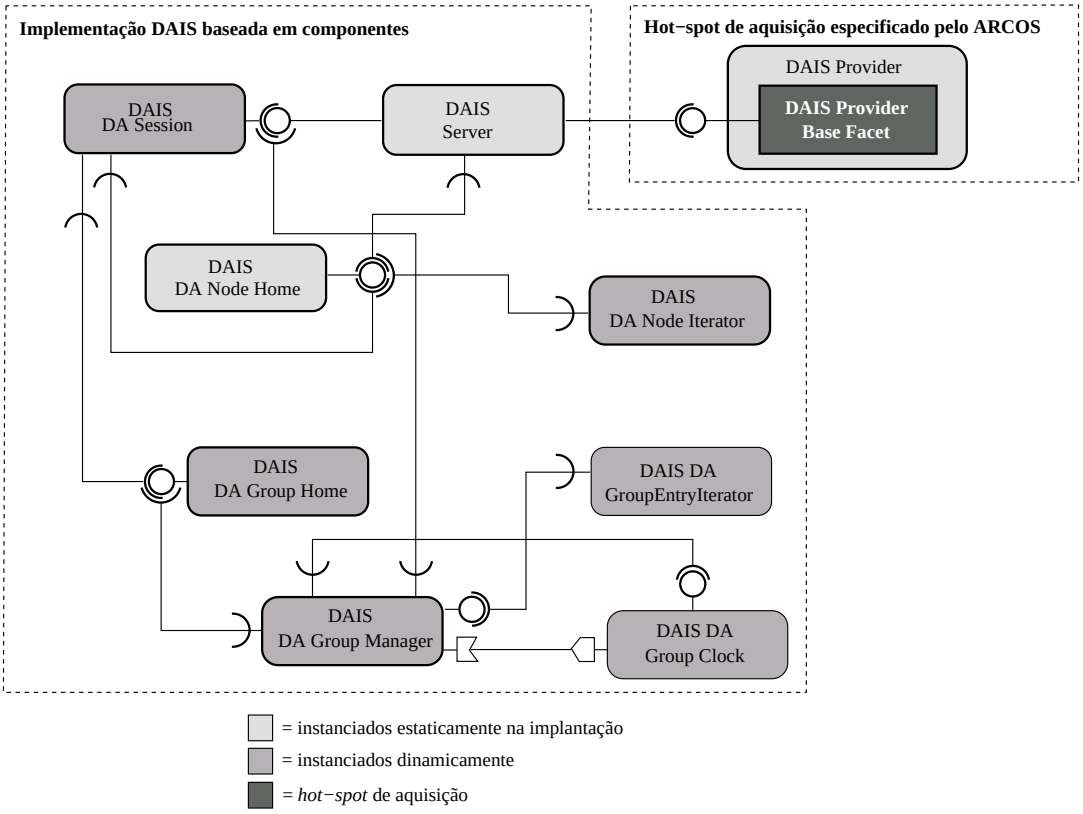


Figura 3.4: Diagrama dos componentes DAIS propostos para o ARCOS

Com o objetivo de disponibilizar uma plataforma totalmente baseada em componentes, o módulo de aquisição de dados do ARCOS constitui uma reestruturação baseada em componentes para os objetos descritos na figura 3.3. Além de promover uma maior modularização e desacoplamento desses objetos, essa reestruturação possibilita a utilização desse servidor DAIS em diversos cenários de aquisição, conforme descrito a seguir. Vale ressaltar que o projeto e implementação do servidor DAIS realizados pelo ARCOS continua compatível com qualquer cliente DAIS em conformidade com o padrão.

A figura 3.4 apresenta a versão baseada em componentes do DAIS, proposta para o ARCOS. Os objetos pertencentes ao grupo "Implementação DAIS baseada em componentes" são aqueles já mencionados na seção 2.6, com exceção do componente *DAISDAGroupClock*. Conforme apresentado anteriormente, ao criar um grupo de aquisição o cliente informa a taxa com a qual os dados deverão ser entregues. Após a definição do grupo e informação da interface de *callback*, o método *on_ data_ change()* desta interface é invocado, pelo servidor DAIS, na taxa informada no momento de criação do grupo. O componente *DAISDAGroupClock* promove uma abstração em relação a qual mecanismo será utilizado para indicar o momento de uma nova coleta. Independente da implementação adotada, o *DAISDAGroupClock* deve gerar eventos assíncronos (não-bloqueantes), na taxa de aquisição especificada para o grupo, destinados ao componente *DAISDAGroupManager*. O *DAISDAGroupManager*, ao receber o evento, realiza os passos necessários para a aquisição e entrega dos dados.

A tabela 3.1 sintetiza os papéis desempenhados por cada um dos componentes da figura 3.4 e apresenta

as justificativas para a criação das conexões entres estes componentes.

3.2.1.1 Hot-spot de aquisição de dados

O grupo "*Hot-spot* de aquisição especificado pelo ARCOS", apresentado na figura 3.4, contém uma interface que não faz parte da especificação DAIS original: a *IDAISProviderBaseFacet*. O *DAIS Provider* é um conceito introduzido pelo ARCOS com o objetivo de generalizar as operações de estruturação e aquisição de dados, possibilitando o uso deste *framework* em uma variedade de situações. Para isso, o ARCOS especifica uma interface denominada *IDAISProviderBaseFacet*, que define métodos para a construção da árvore DAIS e para a aquisição e alteração de uma determinada folha desta árvore.

Ao especializar o ARCOS para uma situação de aquisição de dados particular, o desenvolvedor cria uma interface derivada de *IDAISProviderBaseFacet*, implementa os métodos requeridos para a construção, aquisição e alteração de dados e utiliza esta interface para a definição de uma faceta de um novo componente *DAISProvider*. Esse *provider* específico, representado na figura 3.4 como o componente *DAISProvider* contendo uma faceta implementada pela interface *IDAISProviderBaseFacet*, deve ser capaz de especificar e construir a árvore DAIS e de se comunicar com os dispositivos, de modo a adquirir e alterar os dados em questão.

O código 3.1 apresenta o arquivo IDL especificado pelo ARCOS para a interface *IDAISProviderBaseFacet*. As linhas 16, 18 e 19 definem os três métodos cuja implementação é requerida no desenvolvimento de um *DAIS Provider* específico. O método *build_dais_tree()* é utilizado para a construção da árvore DAIS, indicando quais dados serão disponibilizados pelo *provider* sendo desenvolvido. Uma referência para o objeto *DAISNodeHome* do servidor é enviado para o *DAISProvider*, de modo que este seja capaz de construir a árvore DAIS, através de invocações ao método *add_node()* do *DAISNodeHome*. Durante o processo de aquisição, o método *get_values()* do *DAISProvider* é invocado para cada folha DAIS presente em cada grupo de aquisição criado. O método *get_values()* é invocado periodicamente pelo *DAISDAGroupManager*, com o objetivo de efetivamente adquirir os dados dos dispositivos. O método *set_values()* é invocado pela implementação do método *async_write()* do *DAISDAGroupManager*, com o objetivo de realizar a atualização assíncrona de dados da planta.

O código 3.2 apresenta um exemplo de definição de um *DAIS Provider* específico. A linha 13 define uma interface derivada da interface *IDAISProviderBaseFacet*, disponibilizada pelo ARCOS. A definição dessa nova interface é importante para permitir futuras inclusões de métodos específicos deste *provider*, sem impactos na interface especificada pelo ARCOS. A linha 16 define o componente que representa o novo *DAIS Provider*. Conforme indicado na linha 18, este componente provê uma faceta que implementa os métodos *build_dais_tree()* e *get_value()* (como conseqüência da herança da interface *IDAISProvi-*

Componente	Papel	Componente(s) Conectado(s)	Justificativa da Conexão
DAISServer	Criar sessões de acesso a dados. Definir ponto de conexão para os *DAISProviders*.	*DAISProvider*	Na inicialização do serviço o *DAISProvider* deve ser invocado para realizar a construção da árvore de dados DAIS.
		DAISDANodeHome	Na inicialização do serviço o *DAISDANodeHome* deve ser enviado ao *DAISProvider* para realizar a construção da árvore de dados DAIS.
DAISDA Session	Acompanhar o acesso a dados de um cliente em particular. Disponibilizar os objetos *DAISDANodeHome* e *DAISDAGroupHome*.	*DAISServer*	O componente *DAISDASession* precisa indicar a sua finalização (via execução do método *destroy()*) ao *DAISServer*.
		DAISDANodeHome	O componente *DAISDASession*, no método *node_home()*, deve retornar uma referência para o *singleton DAISDANodeHome*.
		DAISDAGroupHome	O componente *DAISDASession*, no método *group_home()*, deve retornar uma referência para o *DAISDAGroupHome*, específico desta sessão.
DAISDANode Iterator	Possibilitar a entrega gradativa e controlada de nós da árvore DAIS.	*DAISDANodeHome*	O objeto *DAISDANodeIterator* precisa indicar a sua finalização (via execução do método *destroy()*) ao *DAISDANodeHome*.
DAISDAGroup Manager	Gerenciar todas as operações em um grupo de aquisição: informação da taxa, folhas DAIS participantes e a interface de *callback*.	*DAISDAGroupHome*	O objeto *DAISDAGroupManager* precisa indicar a sua finalização (via execução do método *destroy()*) ao *DAISDAGroupHome*.
		DAISServer	O componente *DAISDAGroupManager* precisa obter uma referência para o *DAISProvider* conectado, com o objetivo de realizar operações de escrita de dados.
		DAISDAGroup Clock	O componente *DAISDAGroupManager* precisa realizar as operações de ativação e desativação no *DAISDAGroupClock*.
DAISDAGroup Entry Iterator	Possibilitar a entrega gradativa e controlada de folhas DAIS presentes em grupos de aquisição.	*DAISDAGroupManager*	O componente *DAISDAGroupEntryIterator* precisa indicar a sua finalização (via execução do método *destroy()*) ao *DAISDAGroupManager*.
DAISDAGroup Clock	Abstrair aspectos técnicos para a geração de eventos sinalizadores do período de aquisição.	*DAISDAGroupManager*	O componente *DAISDAGroupClock* precisa indicar, de forma assíncrona, o momento de uma nova coleta de dados.

Tabela 3.1: Sumário dos componentes ARCOS para aquisição de dados

Código 3.1: Arquivo IDL da interface *IDAISProviderBaseFacet* especificada pelo ARCOS

```
#ifndef ARCOSDAISPROVIDERBASE_IDL
#define ARCOSDAISPROVIDERBASE_IDL

#pragma prefix "ufba.br"

#include <Components.idl>
#include <ARCOSDAISDANode.idl>
#include <DAISDAIO.idl>

module ARCOS
{
    module DataAcquisition
    {
        interface IDAISProviderBaseFacet
        {
            void build_dais_tree(in ::ARCOS::DataAcquisition::DAISDANodeHome
                                                                dais_dataaccess_node_home);
            void get_values(in ::DAIS::ItemID node_ids,
                            out ::DAIS::DataAccess::IO::ItemStates item_states);
            ::DAIS::ItemErrors set_values (in ::DAIS::DataAccess::IO::ItemUpdates updates);
        };
    };
};

#endif /* ARCOSDAISPROVIDERBASE_IDL */
```

derBaseFacet). Após a completa implementação do *provider*, essa faceta será devidamente conectada à estrutura interna do ARCOS e seus métodos serão oportunamente invocados para a realização das operações de aquisição. O arquivo IDL apresentado no código 3.2 representa o *DAIS Provider* para comunicação com CLP's via redes *Ethernet*, implementado com o objetivo de validar o *framework* proposto.

Código 3.2: Especificando um novo *DAIS Provider*

```
#ifndef ARCOSDAISETHERNETPLCPROVIDER_IDL
#define ARCOSDAISETHERNETPLCPROVIDER_IDL

#pragma prefix "ufba.br"

#include <Components.idl>
#include <ARCOSDAISProviderBase.idl>

module ARCOS
{
    module DataAcquisition
    {
        interface IEthernetPLCProviderFacet : IDAISProviderBaseFacet
        {
        };
        component DAISEthernetPLCProvider
        {
            provides ::ARCOS::DataAcquisition::IEthernetPLCProviderFacet dais_provider;
            attribute string plc_ip_address;
        };
        home DAISEthernetPLCProviderHome manages DAISEthernetPLCProvider
        {
        };
    };
};

#endif /* ARCOSDAISETHERNETPLCPROVIDER_IDL */
```

O diagrama de seqüência da figura 3.5 apresenta como o ARCOS promove a inversão de controle na

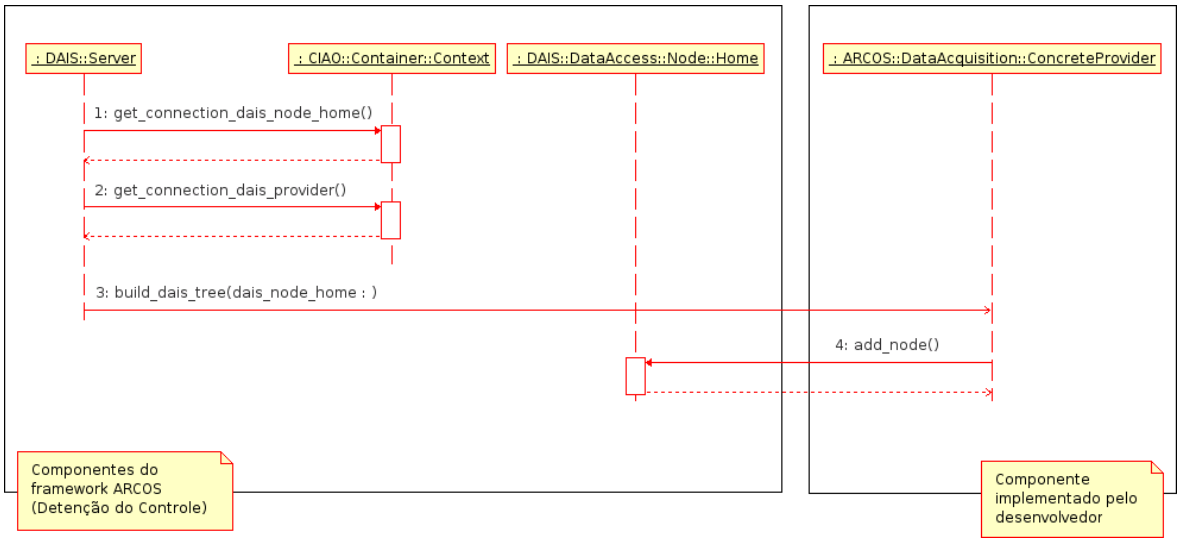


Figura 3.5: Diagrama de seqüência apresentando a inversão de controle realizada pelo ARCOS na construção da árvore DAIS

construção da árvore DAIS. Após a implementação do *DAIS Provider* específico e indicação da conexão deste componente às estruturas internas do ARCOS, todas as operações de consulta e aquisição de dados são realizadas inteiramente pelo *framework*. No momento da inicialização do serviço DAIS, o componente *DAISServer*, implementado pelo ARCOS, obtém a referência dos componente *DAISNodeHome* (passo 1) e *DAISProvider* (passo 2) e invoca o método *build_ dais_ tree()* (passo 3) do *DAISProvider* devidamente conectado, passando o *DAISNodeHome* como parâmetro. A implementação do *provider* específico, realizada pelo desenvolvedor, deve invocar o método *add_ node()* (passo 4) do *DAISNodeHome* recebido como parâmetro, de modo a construir a árvore DAIS desejada.

O diagrama de seqüência da figura 3.6 apresenta como o ARCOS promove a inversão de controle na aquisição dos dados. O componente *DAISDAGroupClock* envia, periodicamente e na freqüência indicada na criação do grupo, um evento indicando que é o momento de realizar uma nova aquisição. O componente *DAGroupManager*, por sua vez, invoca o método *get_ value()* do *DAIS Provider* devidamente conectado, recebendo os dados e os enviando para o Canal de Eventos de Tempo-Real, através da invocação do método *push()*. O componente *DAGroupManager*, ao receber os dados do Canal de Eventos, os repassa para o componente de *callback* (implementado pelo cliente), através da invocação do método *on_ data_ change()*. Esse procedimento se repete periodicamente, na freqüência de aquisição do grupo de dados.

A figura 3.7 resume o projeto do *hot-spot* de aquisição de dados e apresenta com mais detalhes o relacionamento deste módulo com o Serviço de Eventos de Tempo-Real. O bloco denominado "Aquisição de Dados" é uma visão detalhada no bloco apresentado na figura 3.2. O receptáculo *dais_ provider* do componente *DAISServer* (já apresentado na figura 3.4) é um *port* polimórfico, do tipo *ARCOS::DataAcquisition:: IDAISProviderBaseFacet*, que pode ser conectado a qualquer faceta implementada por uma interface derivada de *ARCOS::DataAcquisition::IDAISProviderBaseFacet*. Na figura, o componente *DAISEthernet-PLCProvider*, construído pelo desenvolvedor que utiliza o ARCOS, define uma faceta implementada pela

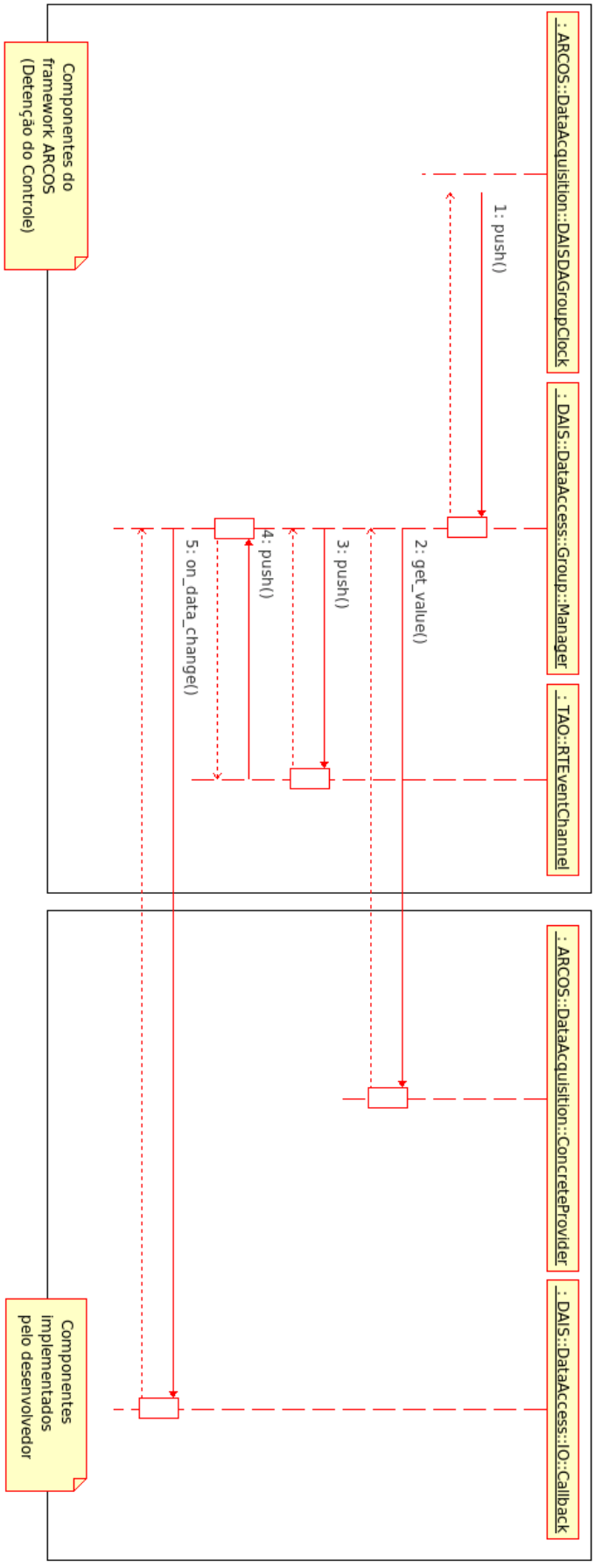


Figura 3.6: Diagrama de seqüência apresentando a inversão de controle realizada pelo ARCOS na aquisição de dados

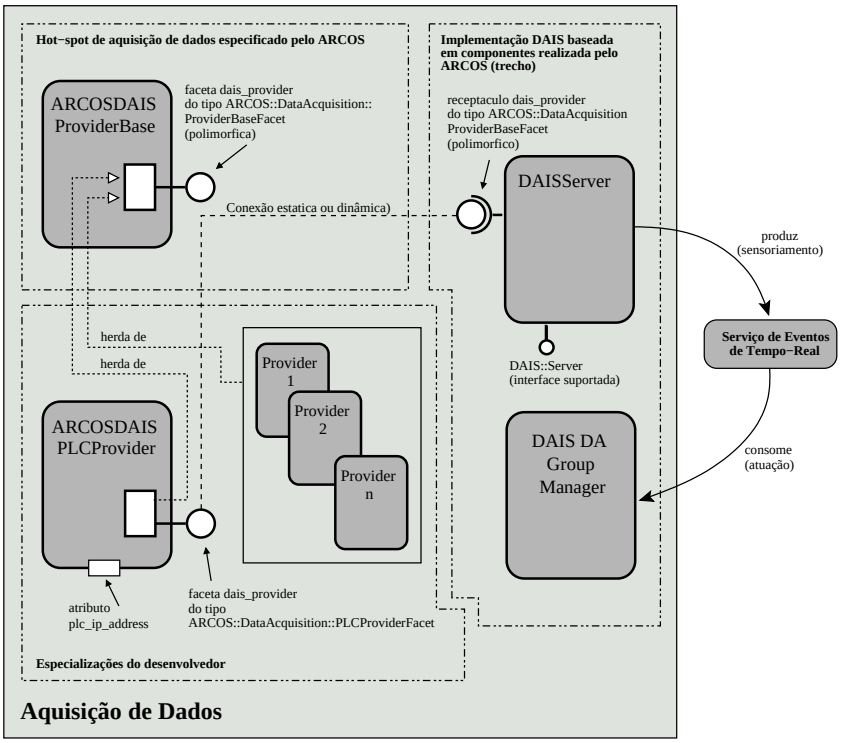


Figura 3.7: Implementação do *hot-spot* de aquisição do ARCOS e as relações do módulo de aquisição com o Serviço de Eventos de Tempo-Real

interface *IEthernetPLCProviderFacet*. Essa interface é derivada da interface *IDAISProviderBaseFacet* e implementa os métodos para definição da árvore DAIS e aquisição/escrita de dados industriais de/-para de um CLP conectado numa rede *Ethernet*. Adicionalmente, esse componente especifica o atributo *plc_ip_address*, tornando possível a indicação, via descritor XML de implantação, o endereço IP do CLP com o qual esse *provider* se relaciona.

Com essa abordagem, toda a especificação e implementação do servidor DAIS do ARCOS pode ser reutilizada em uma série de situações, caracterizadas por diferentes tecnologias de aquisição de dados. O papel do desenvolvedor se limita à construção de um *DAIS Provider* que seja capaz de se comunicar com os dispositivos de aquisição de dados. Ao reutilizar o núcleo funcional do servidor DAIS promove-se um aumento na produtividade e na qualidade das aplicações geradas. De forma semelhante, portanto, podem ser definidos *providers* para acesso via porta paralela, redes industriais, *providers* de interoperabilidade OPC/DAIS, dentre outras tecnologias de aquisição.

Após o desenvolvimento do *DAIS Provider* específico, este deve ser conectado ao componente *DAIS-Server*, apresentado no código 3.3. Na linha 15 é definida a interface *IExtendedDAISServer*. Nas linhas 25 e 26 é definido o receptáculo *dais_provider*, *port* que deverá ser conectado à faceta disponibilizada pelo *DAIS Provider* específico. Conforme apresentado na tabela 3.1 o *DAISServer* também especifica um receptáculo para o componente *DAISDANodeHome*, definido nas linhas 23 e 24, e uma faceta utilizada pelos objetos do tipo *DAISDASession* (linhas 27 e 28).

Código 3.3: Arquivo IDL do componente *DAISServer* especificado pelo ARCOS

```
#ifndef ARCOSDAISSERVER_IDL
#define ARCOSDAISSERVER_IDL

#pragma prefix "ufba.br"

#include <DAISServer.idl>
#include <ARCOSDAISProviderBase.idl>
#include <Components.idl>
#include <ARCOSDAISAccessPoints.idl>

module ARCOS
{
   module DataAcquisition
   {
      interface IExtendedDAISServer : ::DAIS::Server
      {
         void remove_dataaccess_session(in string session_name);
         ::ARCOS::DataAcquisition::IDAISProviderBaseFacet get_dais_provider();
      };

      component DAISServer supports IExtendedDAISServer
      {
         uses ::ARCOS::DataAcquisition::IDAISDANodeHomeAccessPoint
            dais_dataaccess_node_home_ap; // Connection for DAIS tree building
         uses ::ARCOS::DataAcquisition::IDAISProviderBaseFacet
            dais_provider;                // Connection for DAIS data acquisition
         provides IDAISServerAccessPoint
            dais_server_ap;               // Connection for DAIS DA Sessions
      };

      home DAISServerHome manages DAISServer
      {
      };
   };
};

#endif /* ARCOSDAISSERVER_IDL */
```

As conexões do componente *DAISServer*, em especial aquela utilizada para indicar qual *DAIS Provider* será utilizado, são indicadas no arquivo descritor de implantação. O código 3.4 apresenta um trecho desse arquivo descritor, realizando a conexão do *DAISServer* com um *provider* para acesso a CLP's, via redes *Ethernet* (componente *DAISEthernetPLCProvider*).

Código 3.4: Conexão do componente *DAISServer* com um *DAIS Provider* específico

```
   <connection>
      <name>DAISServer_DAISProvider</name>
      <internalEndpoint>
         <portName>dais_provider</portName>
         <kind>Facet</kind>
         <instance>ARCOS-DAISPLCProvider-idd</instance>
      </internalEndpoint>
      <internalEndpoint>
         <portName>dais_provider</portName>
         <kind>SimplexReceptacle</kind>
         <instance>ARCOS-DAISServer-idd</instance>
      </internalEndpoint>
   </connection>
```

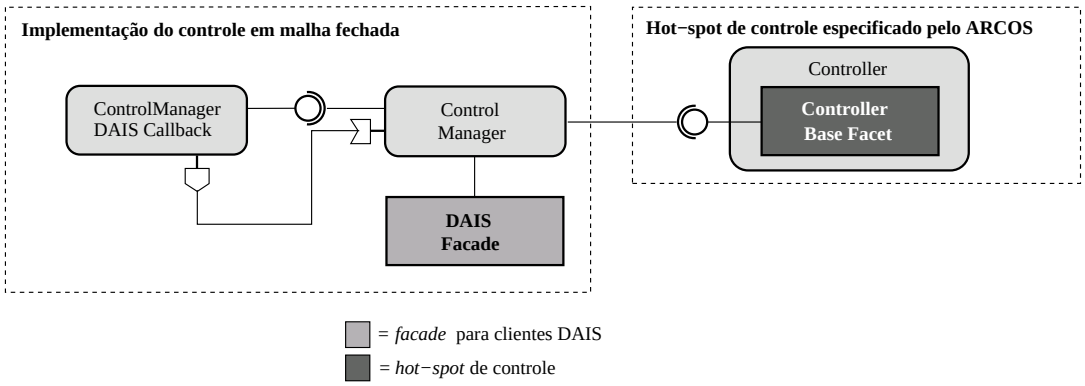


Figura 3.8: Diagrama dos componentes de controle propostos para o ARCOS

3.2.2 Controle

O módulo de controle do ARCOS define um *framework* para implementação de malhas fechadas [35] para controle automático de processos. O objetivo é manter uma determinada variável do processo (temperatura, pressão, vazão etc) em um valor pré-definido pelo operador (*setpoint*). Essa malha fechada é geralmente representada por um ciclo que envolve aquisição de dados de sensoriamento, cálculo da atuação (controle) e a atuação propriamente dita. Os fundamentos matemáticos e as teorias envolvidas na área de controle de processos industriais é um tópico vasto e a discussão e avaliação destes princípios foge ao escopo deste trabalho. Para o leitor interessado em um aprofundamento nessas teorias, detalhes podem ser obtidos em [67, 46].

Objetivando a definição de uma interface genérica para controle, a malha projetada no ARCOS está restrita a um controle discreto de variável única, realizado em um período de amostragem pré-definido. Entretanto, dentro desse escopo de controle, o ARCOS define mecanismos flexíveis que permitem a reutilização dessa malha em conjunto com diferentes estratégias de controle, tais como PID (Proporcional Integral Derivativo), controladores baseados em incertezas (*fuzzy* e *neuro-fuzzy*) e controladores adaptativos.

A figura 3.8 apresenta os componentes para controle projetados pelo ARCOS. Os componentes pertencentes ao grupo "Implementação do controle em malha fechada" são responsáveis pelos processos de aquisição de dados, invocação das operações de controle e atuação na planta. O componente *ControlManager* é o principal responsável pela realização do ciclo de controle. Esse componente é um cliente DAIS de alta prioridade que adquire os dados da planta, invoca a operação de controle do componente *Controller* devidamente conectado, e envia as informações da atuação para o servidor DAIS. O *DAISFacade* é uma classe que facilita a construção de clientes DAIS, sendo utilizada tanto pelo *ControlManager* quanto pelas aplicações supervisórias. Essas aplicações supervisórias e a classe *DAISFacade* são descritas na sub-seção 3.3.5.3. O componente *ControlManagerDAISCallback* representa o objeto de *callback* requerido pelo DAIS para o envio periódico de dados. Ao receber a atualização dos dados na execução do

método *on_data_change()*, o *ControlManagerDAISCallback* envia um evento para o *ControlManager*, sinalizando uma nova execução do *loop* de controle. O componente *Controller*, implementado pelo desenvolvedor, disponibiliza uma faceta implementada pela interface *IControllerBaseFacet*, representando uma estratégia específica de controle. Diversas estratégias de controle podem reutilizar a implementação de aquisição e malhas fechadas do ARCOS, através da utilização do *hot-spot* de controle, descrito a seguir.

3.2.2.1 Hot-spot de controle

Diversas estratégias de controle têm sido propostas continuadamente e com objetivos diversos, tais como a minimização de algumas métricas de controle [35], tais como *overshooting* e tempo de estabilização, e adaptação a variações no ambiente operacional. A existência de uma plataforma flexível de software que permita a implementação e avaliação facilitada de novas estratégias de controle é uma importante demanda atual.

Para isso, o ARCOS especifica uma interface denominada *IControllerBaseFacet*, que define métodos para a realização do cálculo de controle a partir do último valor da variável controlada e do erro atual. Para construir um novo controlador, o desenvolvedor precisa especificar uma interface derivada de *IControllerBaseFacet* e implementar o método *control()*, contendo a estratégia utilizada para realizar o controle. Após essa especificação, deverá ser criado um componente que disponibiliza uma faceta implementada pela interface definida no passo anterior. Essa faceta deverá ser conectada no receptáculo definido no *ControlManager*.

O código 3.5 apresenta o arquivo IDL especificado pelo ARCOS para a interface *IControllerBaseFacet*. A linha 14 especifica o método *control()*, a ser implementado pelo desenvolvedor com o objetivo de definir uma nova estratégia de controle. Esse método deve devolver um valor de atuação em função do erro (diferença entre o valor atual e o valor desejado) e do último valor lido da variável controlada. Diferentes implementações do método *control()* representam diferentes estratégias de controle.

O código 3.6 apresenta um exemplo de definição de um controlador específico. A linha 13 define uma interface derivada da interface *IControllerBaseFacet*, disponibilizada pelo ARCOS. A definição dessa nova interface é importante para permitir futuras inclusões de métodos específicos deste controlador, sem impactos na interface especificada pelo ARCOS. A linha 16 define o componente que representa o novo controlador. Conforme indicado na linha 18, este componente provê uma faceta que implementa o método *control()* (como conseqüência da herança da interface *IControllerBaseFacet*). Após a completa implementação do controlador, essa faceta será devidamente conectada à estrutura interna do ARCOS e seus métodos serão oportunamente invocados para a realização das operações de controle. Esse arquivo IDL representa o controlador PID implementado como objetivo de validar o *framework* proposto. As

Código 3.5: Arquivo IDL da interface *IControllerBaseFacet* especificada pelo ARCOS

```
1  #ifndef ARCOSCONTROLLERBASE_IDL
2  #define ARCOSCONTROLLERBASE_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
7
8  module ARCOS
9  {
10    module Control
11    {
12       interface IControllerBaseFacet
13       {
14          float control (in float error, in float control_old);
15       };
16    };
17 };
18
19 #endif /* ARCOSCONTROLLERBASE_IDL */
```

linhas 19 a 22 representam atributos específicos deste controlador. Estratégias distintas de controle podem ser criadas através da especificação de diferentes componentes e da construção de diferentes montagens, cada uma utilizando a estratégia de controle desejada.

Código 3.6: Especificando um novo controlador

```
1  #ifndef ARCOSPIDCONTROLLER_IDL
2  #define ARCOSPIDCONTROLLER_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
7  #include <ARCOSControllerBase.idl>
8
9  module ARCOS
10 {
11    module Control
12    {
13       interface IPIDControllerFacet : IControllerBaseFacet
14       {
15       };
16       component PIDController
17       {
18          provides ::ARCOS::Control::IPIDControllerFacet controller;
19          attribute float kp;
20          attribute float ki;
21          attribute float kd;
22          attribute unsigned short sampling_rate;
23       };
24       home PIDControllerHome manages PIDController
25       {
26       };
27    };
28 };
29
30 #endif /* ARCOSPIDCONTROLLER_IDL */
```

O diagrama de seqüência da figura 3.9 apresenta como o ARCOS promove a inversão de controle na realização das atividades de controle. O componente *ControlManager* representa um cliente DAIS de alta prioridade que se conecta ao servidor DAIS, cria uma sessão de acesso a dados e define um grupo

de aquisição formado pelo sensor que representa a variável a ser controlada. A aquisição é realizada de forma periódica através do objeto de *callback*.

Para isso, o *ControlManager* adquire a referência do *Callback* devidamente conectado (passo 1) e informa esta referência ao servidor DAIS, através do *DAISFacade* (passo 2). O *DAISFacade* repassa esta referência para o componente *DAISDAGroupManager* do grupo de acesso a dados (passo 3). Após essa configuração inicial e uma vez ativado o Serviço de Eventos de Tempo-Real, o *loop* de controle começa a ser executado, conforme descrito no quadro "Loop de Controle" na figura 3.9.

Ao criar o grupo de aquisição o *ControlManager* informa a freqüência com a qual os dados serão coletados. Nessa mesma freqüência as atividades de controle serão realizadas. O Serviço de Eventos de Tempo-Real envia para o *DAISDAGroupClock*, uma vez por período, uma mensagem indicando a necessidade de uma nova coleta de dados (passo 4), mensagem esta repassada para o *DAISDAGroupManager* (passo 5). O *DAISDAGroupManager* adquire os dados do *DAIS Provider* conectado e envia estes dados através da invocação do método *on_data_change()* do objeto de *callback* do *ControlManager* (passo 6). O objeto de *callback* repassa esses dados para o *ControlManager* através do envio de um evento, realizado pelo método *push_control_data()* (passo 7).

A execução do cálculo do controle é realizado através da invocação do método *control()* do controlador devidamente conectado ao *ControlManager* (passo 8). Esse método devolve a informação de atuação para o *ControlManager* e este solicita a atualização da planta através da invocação do método *async_write()* do *DAISFacade* (passo 9). Finalmente, a atualização é efetivada através da invocação, pelo *DAISFacade*, do método *async_write()* do componente *DAISDAGroupManager* (passo 10). Esse processo se repete initerruptamente, a cada período informado no grupo de aquisição que representa o sensor da variável controlada.

A figura 3.10 resume o projeto do *hot-spot* de controle e apresenta com mais detalhes o relacionamento deste módulo com o Serviço de Eventos de Tempo-Real. O bloco denominado "Controle" é uma visão detalhada do bloco apresentado na figura 3.2. O principal componente desse módulo é o *ControlManager*, responsável pela implementação do ciclo de controle e pela delegação dos cálculos de atuação para o componente *Controller* devidamente conectado. O *ControlManager* é um cliente DAIS de alta prioridade, que define um grupo de aquisição de dados formado pela folha DAIS que representa a variável a ser controlada (temperatura, pressão, vazão etc). Esse grupo realiza a aquisição de dados com a freqüência indicada, através do descritor XML de implantação, no atributo *sampling_rate* do componente *ControlManager*. Após a aquisição de dados, em cada período, o *ControlManager* invoca o método *control()* do componente *Controller* devidamente conectado, obtendo as informações de atuação que serão enviadas de volta ao módulo de aquisição. Essas informações de atuação serão consumidas pelo objeto *DAISDAGroupManager* correspondente ao grupo de aquisição do *ControlManager* e refletidas nos dispositivos da planta, realizando a operação de controle. Além do atributo *sampling_rate*, o *ControlManager* disponibiliza atributos

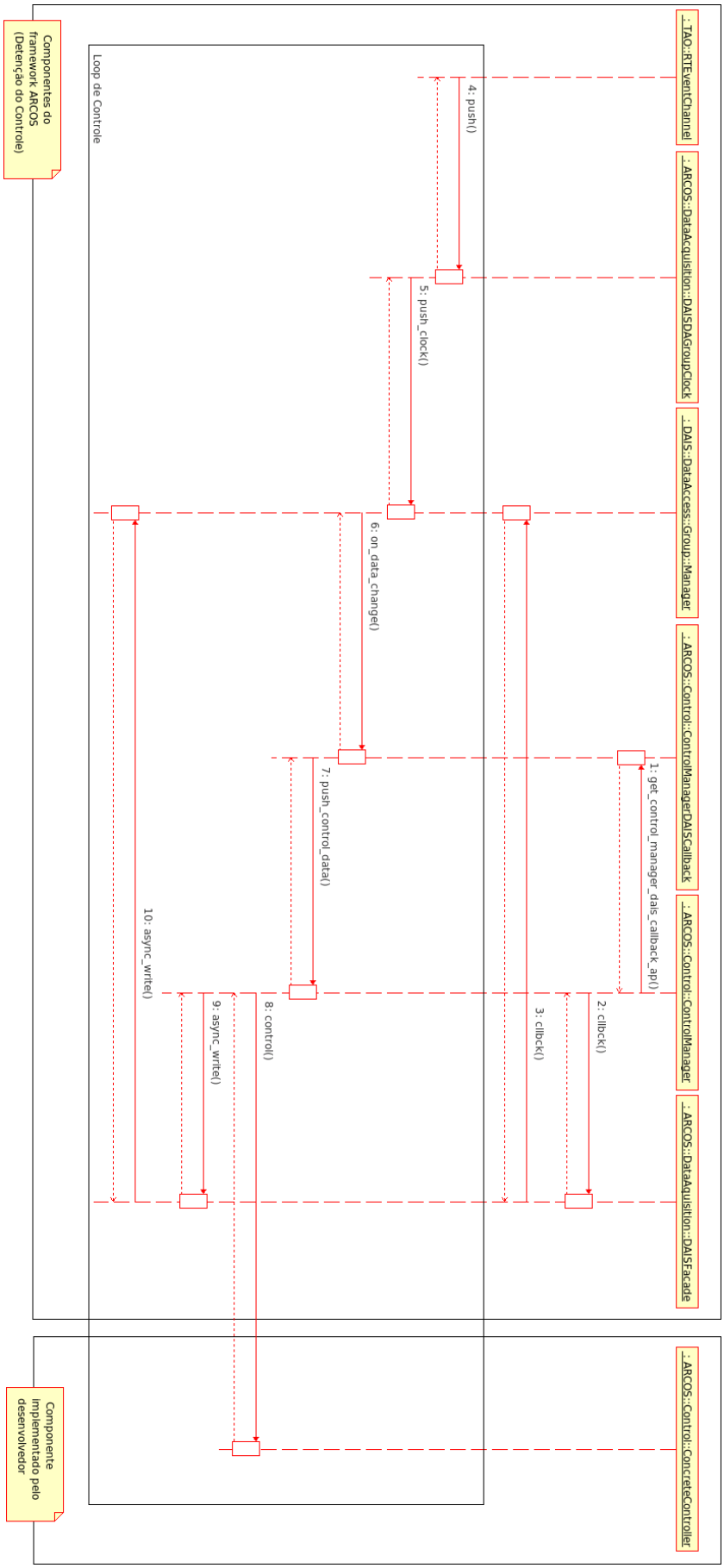


Figura 3.9: Diagrama de seqüência apresentando a inversão de controle realizada pelo ARCOS na realização das atividades de controle

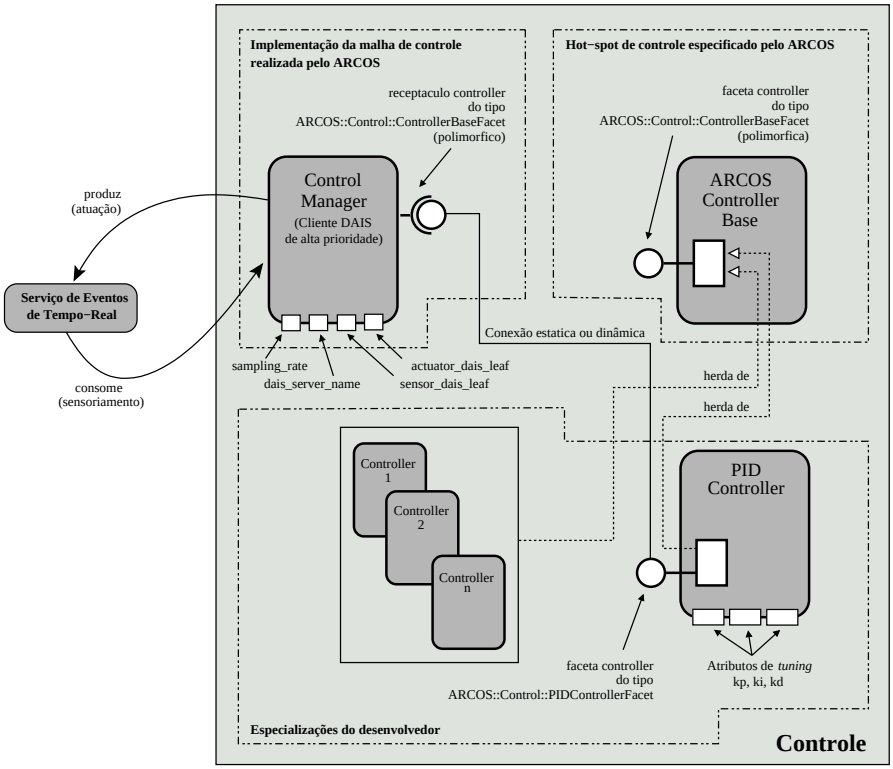


Figura 3.10: Implementação do *hot-spot* de controle do ARCOS e as relações do módulo de controle com o Serviço de Eventos de Tempo-Real

para indicação de nome do servidor DAIS a ser utilizado (*dais_server_name*) e das folhas DAIS que representam o sensor da variável controlada (*sensor_dais_leaf*) e do atuador (*actuator_dais_leaf*). Todos esses atributos são configurados em tempo de implantação, através do descritor XML ou modificados dinamicamente, através dos mecanismos de reimplantação e reconfiguração do ReDaC.

Com essa abordagem, toda a especificação e implementação do *loop* de controle do ARCOS pode ser reutilizada em uma série de situações, caracterizadas por diferentes estratégias de controle. O papel do desenvolvedor se limita à construção de um controlador que implementa o método *control()* com a estratégia desejada.

Após o desenvolvimento do controlador específico, este deve ser conectado ao componente *ControlManager*, apresentado no código 3.7. As linhas 14 a 18 definem a interface *IControlManager*, responsável pela implementação dos métodos para ativação e desativação do *loop* de controle. Na linha 22 é definido o receptáculo *controller*, *port* que deverá ser conectado à faceta disponibilizada pelo controlador específico, implementado pelo desenvolvedor. As linhas 23 e 24 definem o receptáculo *control_manager_dais_callback_ap*, utilizado para aquisição do objeto de *callback* devidamente conectado e que será utilizado para a recepção dos dados da planta. A linha 25 define o depósito de eventos do tipo *ControlData*, enviados pelo *callback* para sinalizar o momento de uma nova ação de controle. As linhas 26 a 31 definem atributos de configuração do *ControlManager* para indicar a taxa de amostragem do controlador (*sampling_rate*), o servidor DAIS a ser utilizado (*dais_server_name*), os identificadores das

folhas DAIS correspondentes ao sensor (*sensor_dais_leaf*) e ao atuador (*actuator_dais_leaf*), o nome que será utilizado na sessão DAIS que representa o controlador (*RegisterNaming*) e o *setpoint* especificado (*setpoint*).

Código 3.7: Arquivo IDL do componente *ControlManager* especificado pelo ARCOS

```
#ifndef ARCOSCONTROLMANAGER_IDL
#define ARCOSCONTROLMANAGER_IDL

#pragma prefix "ufba.br"

#include <ARCOSControllerBase.idl>
#include <ARCOSControlManagerDAISCallback.idl>
#include <DAFIdentifiers.idl>
#include <ControlDataEvent.idl>
#include <Components.idl>

module ARCOS {
   module Control {
      interface IControlManager
      {
         void activate();
         void deactivate();
      };

      component ControlManager supports IControlManager
      {
          uses ::ARCOS::Control::IControllerBaseFacet controller;
          uses ::ARCOS::Control::IControlManagerDAISCallbackAccessPoint
                                  control_manager_dais_callback_ap;
          consumes EControlData                    control_data;
          attribute unsigned short                 sampling_rate;
          attribute string                         dais_server_name;
          attribute ::DAFIdentifiers::ResourceID   sensor_dais_leaf;
          attribute ::DAFIdentifiers::ResourceID   actuator_dais_leaf;
          attribute string                         RegisterNaming;
          attribute float                          setpoint;
      };

      home ControlManagerHome manages ControlManager
      {
      };
   };
};

#endif /* ARCOSCONTROLMANAGER_IDL */
```

As conexões do componente *ControlManager*, em especial aquela utilizada para indicar qual controlador será utilizado, são indicadas no arquivo descritor de implantação. O código 3.8 apresenta um trecho deste arquivo descritor, realizando a conexão do *ControlManager* com um controlador PID convencional.

3.2.3 Supervisão

Nos sistemas de S&C, as atividades de monitoração do processo industrial são realizados pelos chamados sistemas supervisórios, ou ainda, sistemas SCADA (*Supervisory Control And Data Acquisition*). Através desses sistemas é possível monitorar o estado de sensores e atuadores, bem como realizar operações tais como mudança de valores de *setpoints*, armazenamento de históricos em banco de dados, definição de políticas de segurança relacionadas à utilização do sistema, dentre outras.

Código 3.8: Conexão do componente *ControlManager* com um controlador específico

```
<connection>
    <name>ControlManager_PIDController</name>
    <internalEndpoint>
        <portName>controller</portName>
        <kind>Facet</kind>
        <instance>ARCOS-PIDController-idd</instance>
    </internalEndpoint>
    <internalEndpoint>
        <portName>controller</portName>
        <kind>SimplexReceptacle</kind>
        <instance>ARCOS-ControlManager-idd</instance>
    </internalEndpoint>
</connection>
```

Dentre as principais características atualmente requeridas em tais sistemas, destaca-se: interoperabilidade, previsibilidade e integração. Sistemas SCADA devem ser capazes de adquirir e visualizar dados oriundos de diversos dispositivos de aquisição, tais como CLP's, sensores inteligentes conectados numa rede industrial, porta paralela ou porta USB (*Universal Serial Bus*). A possibilidade de utilização de diversos protocolos padronizados de aquisição de dados, tais como o DAIS ou OPC, também é uma característica desejável em tais sistemas. A supervisão de algumas plantas industriais pode requerer que o estado apresentado no sistema SCADA seja uma representação fiel e atual do chão-de-fábrica, conduzindo a restrições temporais que devem ser atendidas e que geram impactos no projeto e desenvolvimento desses sistemas. Atualmente, a possibilidade de integração dos sistemas de supervisão com ambientes tais como a Internet representa uma forte tendência. Com essa abordagem, a supervisão de uma planta industrial pode ser realizada a partir de qualquer *browser* web ou dispositivos móveis.

Nesse contexto, o ARCOS disponibiliza algumas aplicações com funcionalidades de supervisão, reunidas numa ferramenta denominada *ARCOS Management Tool* (AMT) (figura 3.11). O AMT é formado por três aplicações principais: o *DAIS Server Browser*, o *DAIS Server Manager* e o *ARCOS Assembly Tool*. O apêndice G contém o manual de utilização do AMT.

3.2.3.1 DAIS Server Browser

O padrão DAIS especifica um conjunto de interfaces CORBA para obtenção da árvore DAIS que representa os dados disponibilizados pelo servidor e para a conseqüente aquisição destes dados. O *DAIS Server Browser*, disponibilizado pelo ARCOS, é um cliente DAIS em conformidade com o padrão. Através dessa aplicação é possível visualizar a árvore DAIS e adquirir dados de qualquer servidor DAIS, seja este o servidor disponibilizado pelo ARCOS ou um outro servidor implementado numa outra linguagem de programação e executado em um outro sistema operacional. Para isso, precisa-se que esse servidor esteja em conformidade com o padrão DAIS e tenha sua referência registrada em um serviço de nomes que siga o padrão do OMG (*COS Naming Service*).

Dentre as funcionalidades disponibilizadas pelo *DAIS Server Browser* pode-se destacar:

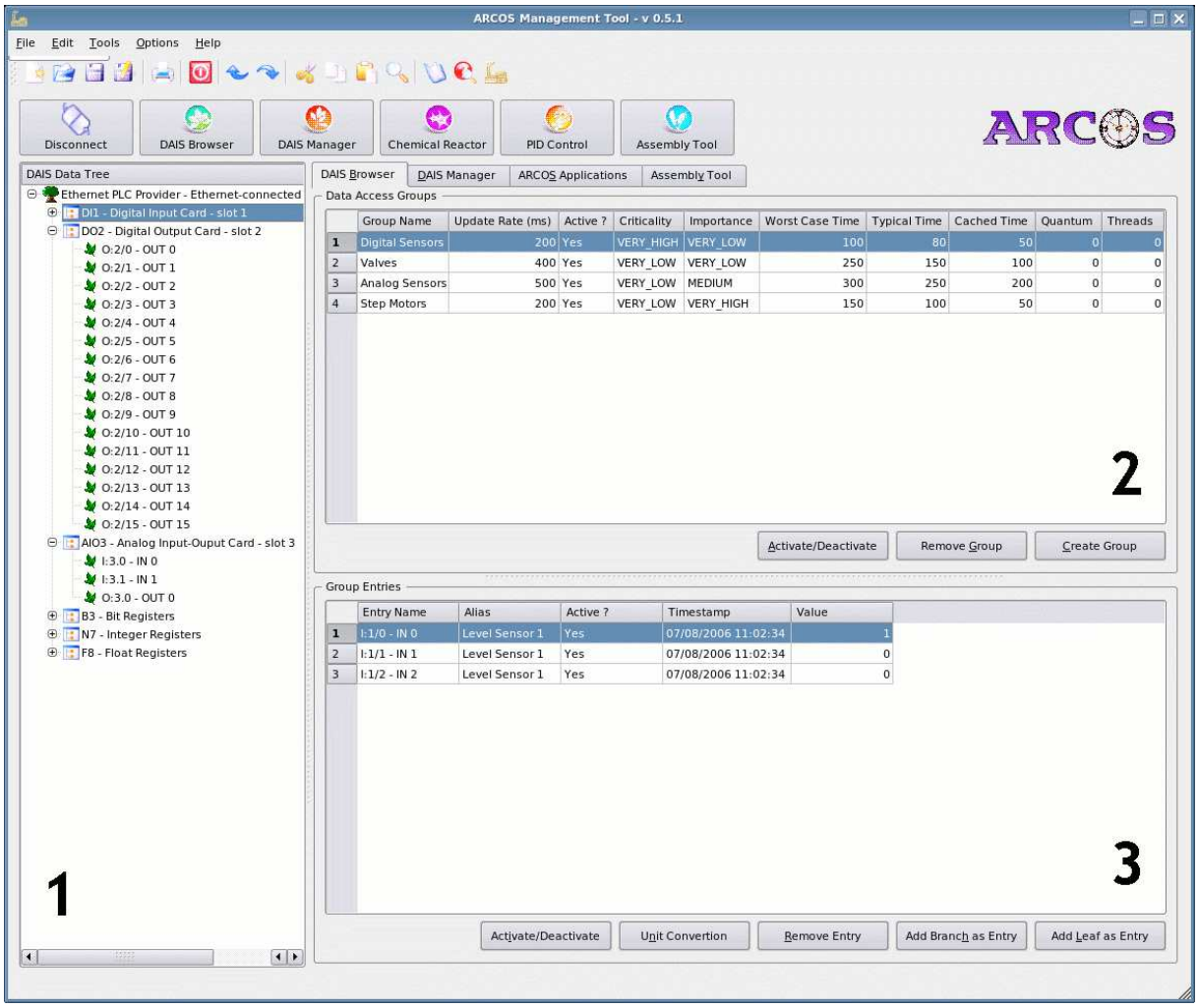


Figura 3.11: *ARCOS Management Tool*

- Visualização do Servidor de Nomes: o primeiro passo para a conexão a um servidor DAIS consiste na aquisição de uma referência remota para este servidor. Uma das formas de aquisição dessa referência consiste na utilização de um Servidor de Nomes. Na inicialização do servidor DAIS, este deve ser registrado no Servidor de Nomes, através de uma operação de *bind* entre o nome atribuído ao servidor e o componente que o representa. Ao utilizar o CIAO e o DAnCE, essa operação de *bind* é realizada automaticamente, durante a operação de implantação, poupando o desenvolvedor da realização deste trabalho. Uma vez publicada essa referência, clientes interessados na utilização do serviço devem realizar uma consulta ao Servidor de Nomes, solicitando a referência do servidor DAIS em questão. O visualizador do Servidor de Nomes, disponibilizado pelo *DAIS Server Browser*, permite a visualização de quais servidores DAIS estão disponíveis no momento, e a escolha de com qual servidor deverá ser realizada a conexão. A figura 3.12 apresenta o visualizador do Servidor de Nomes. Esse visualizador pode ser utilizado em conjunto com um Servidor de Nomes implementado em qualquer linguagem de programação, desde que esteja em conformidade com o padrão estabelecido pelo OMG (*COS Naming Service*).

- Visualização da árvore DAIS disponibilizada pelo servidor: após a escolha do servidor DAIS a ser utilizado, o *DAIS Server Browser* realiza a consulta recursiva da árvore DAIS disponibilizada. A estruturação e conteúdo dessa árvore são definidos pelo servidor DAIS e, mais especificamente, pelo *DAIS Provider* utilizado. O *DAIS Server Browser* disponibiliza as seguintes funcionalidades: exibição da árvore DAIS; expansão e contração de ramos; e seleção de folhas e ramos para inclusão em grupos de aquisição. A região 1 apresentada na figura 3.11 apresenta a árvore DAIS coletada de um servidor DAIS capaz de obter dados de um CLP conectado a uma rede *Ethernet*.

- Criação de grupos de aquisição de dados: conforme indicado nas seções passadas, a unidade de aquisição de dados no DAIS é chamada de grupo. Um grupo é formado por uma coleção de folhas da árvore DAIS do servidor e são criados livremente pelo usuário. Desta forma, pode-se criar, por exemplo, um grupo para sensores analógicos, um grupo para válvulas digitais ou outro grupo para motores. Para cada grupo deve ser indicado qual a freqüência de aquisição de dados (período). Todas essas operações podem ser realizadas, de forma visual, através do *DAIS Server Browser*. Uma vez criado um grupo, o usuário pode inserir folhas individuais da árvore ou todas as folhas pertencentes a um determinado ramo. As regiões 2 e 3 apresentadas na figura 3.11 apresentam os painéis para criação de grupos e inclusão de folhas em grupos, respectivamente.

- Monitoração on-line dos dados coletados: após a criação dos grupos desejados e a inclusão das folhas DAIS o processo de aquisição pode ser iniciado, através da ativação do Serviço de Eventos de Tempo-Real. As alterações ocorridas no chão-de-fábrica são instantaneamente exibidas no painel de visualização de grupos do *DAIS Server Browser*. O Serviço de Eventos de Tempo-Real do TAO se encarrega de executar os testes de escalonabilidade, indicando se todos os clientes DAIS podem ter seus requisitos temporais atendidos, e de realizar o *dispatch* dos dados coletados segundo as prioridades definidas pelo Serviço de Escalonamento.

O ARCOS disponibiliza duas versões do *DAIS Server Browser*: versão *desktop* (apresentada na figura 3.11) e versão web (apresentada na figura 3.13). A versão web apresenta as mesmas funcionalidades da versão *desktop*, com a exceção que a aquisição é atualizada através de um botão de *refresh*, disponibilizado na ferramenta.

O *DAIS Server Browser* representa, desta forma, um cliente DAIS passível de ser utilizado com qualquer servidor DAIS em conformidade com o padrão. Devido ao fato de o Serviço de Eventos de Tempo-Real realizar todas as operações de testes de escalonabilidade e determinação de prioridades de forma *off-line*, a criação de novos grupos e a reconfiguração das folhas DAIS que compõem um grupo não podem ser realizados após a ativação do Serviço de Eventos. Essa restrição pode ser superada à medida em que novas políticas para avaliação da escalonabilidade e determinação de prioridades são introduzidas

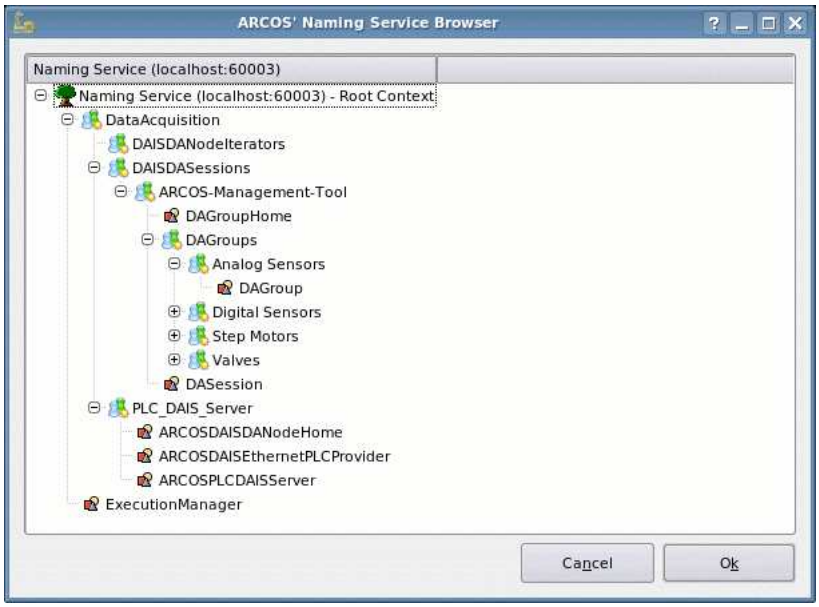


Figura 3.12: Visualizador do Servidor de Nomes, disponibilizado pelo *ARCOS Management Tool*

no Serviço de Eventos de Tempo-Real.

3.2.3.2 DAIS Server Manager

O *DAIS Server Manager* permite a monitoração de um servidor DAIS específico, visualizando as sessões atualmente criadas e os grupos de aquisição criados em cada sessão. Enquanto o *DAIS Server Browser* apresenta funcionalidades para utilização do serviço de aquisição, o *DAIS Server Manager* constitui uma aplicação para monitoração do estado geral do serviço. A figura 3.14 ilustra o *DAIS Server Manager* apresentando as sessões DAIS e os grupos de aquisição criados em cada sessão. Além da monitoração de um servidor DAIS específico, o *DAIS Server Manager* representa o local onde é realizada a ativação do Serviço de Eventos de Tempo-Real. Após a ativação desse serviço, o processo de aquisição de dados se inicia para todos os clientes DAIS conectados no servidor.

3.3 A implementação

Nesta seção serão apresentados aspectos técnicos relacionados à implementação dos módulos descritos nas seções anteriores. O ARCOS é um *framework* para desenvolvimento de sistemas de S&C baseado em três tecnologias já apresentadas neste texto: o ACE, o TAO e o CIAO. A sub-seção 3.3.1 apresenta os requisitos de software necessários à implementação e compilação do ARCOS. Devido ao fato de ser uma especificação consideravelmente recente (junho de 2002), o CCM conta, atualmente, com implementações parciais, que não disponibilizam ainda todos os recursos oferecidos pela especificação. Desta forma, o desenvolvimento do ARCOS foi realizado concomitantemente ao desenvolvimento do CIAO e o histórico

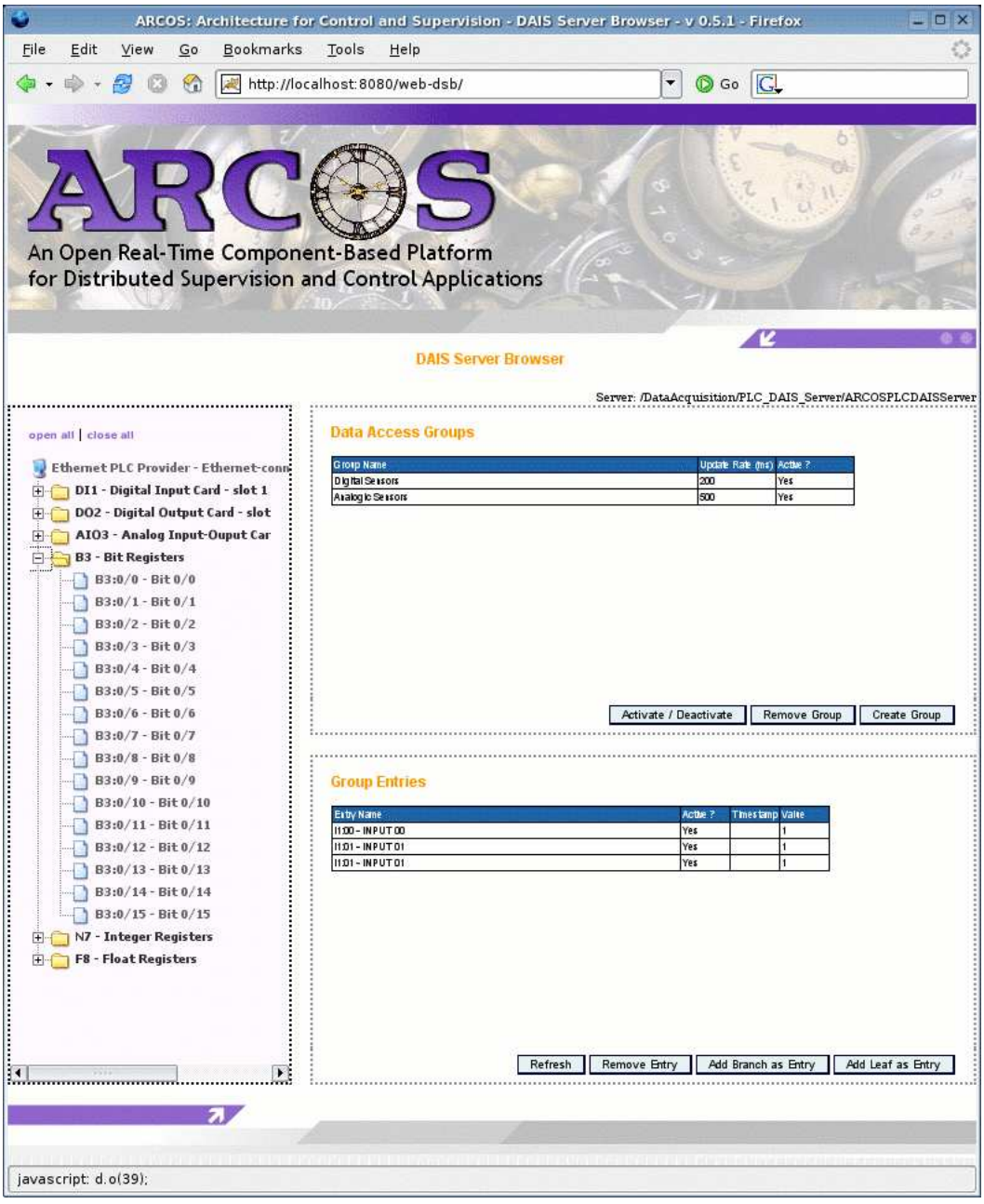


Figura 3.13: Versão web do *DAIS Server Browser*

desse desenvolvimento é apresentado na sub-seção 3.3.2. A sub-seção 3.3.3 apresenta a solução utilizada atualmente no ARCOS para a criação e reconfiguração dinâmica de novas instâncias de componentes. A sub-seção 3.3.4 apresenta a ferramenta disponibilizada pelo CIAO para compilação de projetos que envolvem um número considerável de classes (o MPC), enquanto as seções 3.3.5.1, 3.3.5.2 e 3.3.5.3 apresentam aspectos importantes da implementação dos módulos de aquisição, controle e supervisão, respectivamente.

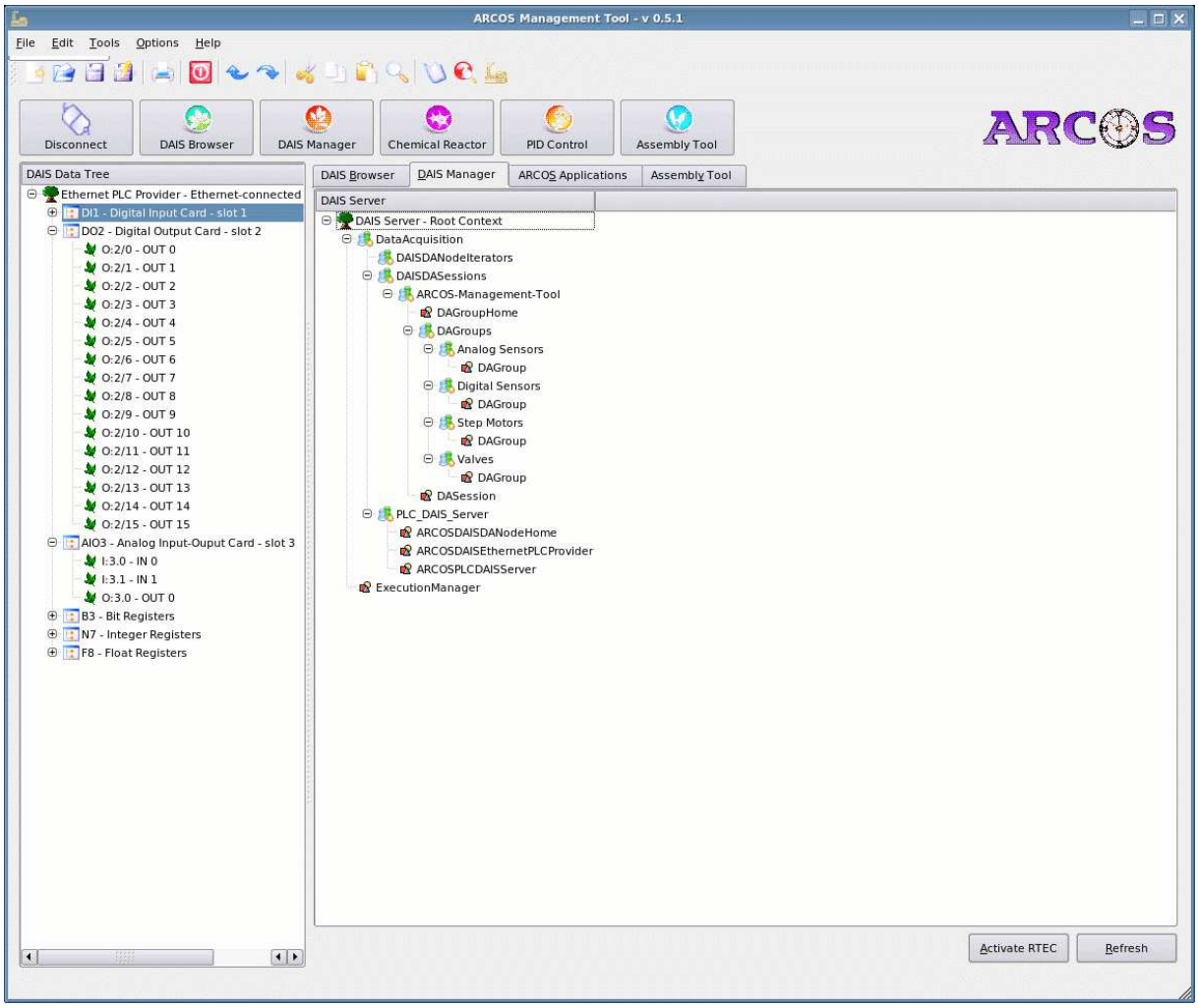


Figura 3.14: *DAIS Server Manager* apresentando as sessões e os grupos de cada sessão

3.3.1 Requisitos de software

O ARCOS foi projetado e desenvolvido através da utilização de ferramentas livres e, portanto, não depende de soluções proprietárias, sejam relacionadas a ferramentas ou protocolos. A plataforma foi desenvolvida em um ambiente formado pelo sistema operacional GNU/Linux [27], utilizando a linguagem de programação Standard C++ [98] e suas ferramentas auxiliares (*cpp*, *libtool*, *make* etc). Os softwares requeridos para a completa compilação do ARCOS são:

- Compilador g++ 4.0.3 e ferramentas auxiliares: além do compilador g++, ferramentas comumente utilizadas no desenvolvimento de sistemas C++ são requeridas, tais como: o preprocessador C (*cpp*), ferramenta para construção de bibliotecas (*libtool*), ferramenta de gerenciamento de compilações (*make*), ferramentas de depuração (*gdb* e *ldd*) e ferramentas para geração de documentação (*doxygen*).

- ACE 5.5.1, TAO 1.5.1 e CIAO 0.5.1 (*Beta*): devido ao fato de o CIAO ser um projeto em andamento

e com constantes liberações de novas versões, recomenda-se a utilização das versões aqui indicadas para o ACE, TAO e CIAO. O ARCOS seguirá a mesma numeração de versão utilizada no CIAO, ou seja, o ARCOS 0.5.1 deverá ser compilado com o CIAO 0.5.1 e assim sucessivamente. O ACE, TAO e CIAO podem ser obtidos no site *http://deuce.doc.wustl.edu/Download.html* e estão disponíveis nas versões *Release*, *BFO* (*Bug Fix Only*) e *Beta*. A versão *Release* representa a versão mais estável porém não apresenta uma série de funcionalidades recentemente implementadas. A versão *BFO* consiste na versão *Release* aprimorada somente com correção de *bugs*, porém ainda sem a presença de novas funcionalidades. A versão *Beta* representa a última liberação e contém as funcionalidades recentemente implementadas. Apesar de recentemente implementada, a versão *Beta* não tem apresentado problemas quanto à compilação e execução dos principais serviços. Versões com atualizações diárias podem ser obtidas através do CVS (*Concurrent Versioning System*) do *DOC Group*. Instruções para compilação e instalação do ACE, TAO e CIAO estão disponíveis nos arquivos ACE-INSTALL, TAO-INSTALL e CIAO-INSTALL, presentes no pacote obtido do *site* do *DOC Group*. No projeto do ARCOS foi utilizada a versão *Beta* 0.5.1.

- KDevelop 3.2.2: o KDevelop [40] é um projeto iniciado em 1998 com o intuito de disponibilizar uma IDE (*Integrated Development Environment*) para o desenvolvimento de aplicações voltadas para o sistema operacional GNU/Linux. Dentre as suas principais funcionalidades pode-se destacar: facilidades para edição, compilação e depuração de sistemas; projeto visual de interfaces de usuário e ferramentas para documentação, distribuição e internacionalização (interfaces de usuários em vários idiomas). O KDevelop 3.2.2 foi a IDE utilizada neste projeto para o desenvolvimento do *ARCOS Management Tool*.

- Qt 3: o Qt [101] é um *toolkit* para desenvolvimento de interfaces gráficas amplamente utilizado na construção do KDE (*The K Desktop Environment*) [39]. Além de disponibilizar os recursos básicos para projeto de interfaces (botões, caixa de texto, caixas de seleção etc), o Qt também oferece uma série de soluções para manipulação de dados multimídia (imagens, vídeos e áudio), desenvolvimento multi-tarefa e internacionalização. O Qt foi o *toolkit* utilizado para o desenvolvimento das interfaces do *ARCOS Management Tool*.

Apesar de ter sido implementado e testado numa plataforma baseada no sistema operacional GNU/Linux, o ARCOS faz uso de bibliotecas portáveis, com utilização comprovada em outros ambientes tais como o sistema operacional Microsoft Windows. O ACE oferece uma boa portabilidade e tem sido utilizado com sucesso em plataformas tais como Microsoft Windows, GNU/Linux, Solaris, SGI IRIX, MAC OS X e sistemas operacionais de tempo-real, tais como LynxOS, VxWorks e QnX Neutrino. O TAO, CIAO e ARCOS, devido ao fato de serem baseados no ACE também são passíveis de serem portados para outras

plataformas. O Qt está livremente disponível em versões para o GNU/Linux e Microsoft Windows.

3.3.2 Histórico

O desenvolvimento de sistemas baseados em componentes apresenta uma série de vantagens que facilitam o desenvolvimento de aplicações flexíveis, reutilizáveis e escaláveis. Dentre essas vantagens, destaca-se a presença de um ambiente de software responsável para execução de componentes e gerência dos recursos e serviços utilizados por estes componentes. Esse ambiente de software, denominado *container*, poupa o desenvolvedor da implementação de tarefas recorrentes em sistemas distribuídos, tais como: registro de componentes no servidor de nomes, utilização de serviços não-funcionais (persistência, segurança, eventos etc) e gerenciamento de recursos (conexões com banco de dados e instâncias de componentes).

Uma tarefa comum em sistemas baseados em componentes distribuídos é a necessidade de criação e remoção de novas instâncias de componentes, com o objetivo de atender requisições de clientes que utilizam o serviço. Especificamente no projeto do ARCOS, nota-se que o DAIS apresenta um comportamento bastante dinâmico, requerendo constantemente a criação e remoção de componentes que representam, por exemplo, sessões, grupos de dados e *iterators*.

Nesse contexto, algumas abordagens distintas para a realização dessa tarefa foram avaliadas no desenvolvimento do ARCOS. Essas abordagens foram utilizadas em diferentes estágios do desenvolvimento do CIAO, caracterizando a funcionalidade disponível no momento. À medida em que novas soluções, mais fáceis de serem utilizadas e com benefícios ligados à flexibilidade e escalabilidade, foram implementadas no CIAO, o ARCOS foi reprojetado de modo a utilizar estas tecnologias.

As seções a seguir apresentam as três abordagens avaliadas, ao longo do desenvolvimento do ARCOS, para a criação, remoção e reconfiguração de instâncias de componentes: utilização de objetos CORBA 2.x, utilização de componentes *home* e utilização do ReDaC.

3.3.2.1 1ª solução: utilização de objetos CORBA 2.x

O CORBA 2.x, apesar de todos os benefícios relacionados à independência das tecnologias envolvidas, apresenta como principal desvantagem o grande esforço e o conhecimento especializado necessários para o desenvolvimento de aplicações distribuídas. Nessa abordagem, o desenvolvimento desses sistemas requer a execução, pelo desenvolvedor, das seguintes tarefas:

1. Definição do arquivo IDL do objeto: nesse arquivo são especificadas, para cada objeto, as interfaces que cada um implementa. É necessário que o desenvolvedor possua um bom conhecimento sobre os recursos disponíveis na linguagem IDL e como estes recursos serão mapeados na linguagem alvo

utilizada no projeto.

2. Compilação dos arquivos IDL: nessa fase serão gerados os *skeletons* e *stubs* que serão utilizados na construção do *servant* e do cliente, respectivamente.

3. Implementação do *servant*: nessa fase o desenvolvedor deverá implementar a classe que deriva do *skeleton* gerado e que implementa os métodos presentes na interface definida no arquivo IDL. É necessário um bom conhecimento a respeito do mapeamento realizado entre a linguagem IDL e a linguagem alvo do projeto.

4. Implementação do servidor: no CORBA 2.x, a implementação do servidor que disponibiliza os objetos distribuídos é de responsabilidade do desenvolvedor, não existindo a presença de um *container* genérico para execução de objetos. Essa implementação requer um conhecimento aprofundado sobre os serviços e mecanismos do CORBA para a construção de aplicações que sejam flexíveis e escaláveis. A construção desse servidor envolve a implementação das seguintes tarefas.

   (a) Inicialização do ORB: parâmetros tais como a localização (endereço IP e porta) do Servidor de Nomes e a localização onde o objeto distribuído será disponibilizado podem ser ajustados no momento da inicialização do ORB (método *init()* do objeto ORB).

   (b) Aquisição da referência para o Servidor de Nomes: a referência para o Servidor de Nomes deve ser obtido através da invocação do método *resolve_initial_references()* do objeto ORB. Essa referência será utilizada no momento do registro do objeto, conforme descrito a seguir.

   (c) Criação e configuração do Adaptador de Objetos: no CORBA 2.x a definição das políticas relacionadas a fatores tais como escalabilidade, persistência e segurança é um trabalho a ser totalmente realizado pelo desenvolvedor. Um *servant*, após instanciado, deve ser ativado em uma entidade denominada Adaptador de Objetos. Esse Adaptador de Objetos se torna responsável pelo gerenciamento do ciclo de vida desse *servant*, possibilitando a implementação de operações tais como persistência do estado do *servant* e o mapeamento de requisições de clientes em *servants*. O CORBA 2.x faz uma distinção em objeto distribuído e *servant*. Um objeto distribuído é uma visão utilizada pelo cliente para a invocação de operações remotas e não implica numa relação um-para-um com *servants*. Um mesmo *servant* pode atender requisições de diversos clientes, contribuindo, desta forma, para a escalabilidade e gerência dos recursos do sistema. A instanciação de *servants*, criação e configuração das políticas dos Adaptadores de Objetos e a ativação de *servants* nestes adaptadores é total responsabilidade do desenvolvedor, exigindo experiência e conhecimento especializado para o desenvolvimento dessas aplicações.

(d) Registro do objeto distribuído no Servidor de Nomes: para que clientes adquiram referências para o objeto distribuído, este deve ter seu endereço registrado no Servidor de Nomes, através da invocação do método *bind()*, disponibilizado por este servidor. A estrutura interna do Servidor de Nomes é representada através de uma árvore. Os nós dessa árvore são chamados de contextos e as folhas são referências para objetos distribuídos. Desta forma, o serviço pode ser organizado através da criação de novos contextos. Por exemplo, pode-se criar um contexto que agrupa objetos de aquisição de dados e outro contexto que agrupa objetos de controle. Todas as operações de criação de contextos e registro de objetos distribuídos devem ser implementadas pelo desenvolvedor.

(e) Ativação do ORB: após a realização de todas as operações de configuração de adaptadores, ativações de *servants* e registro no Servidor de Nomes, o objeto ORB é ativado através da execução do método *run()*, disponibilizado por este objeto. A partir desse momento, o *servant* está disponível para tratamento de invocações remotas.

5. Implementação do cliente: um cliente CORBA 2.x é geralmente formado pela execução dos seguintes passos: inicialização do ORB, aquisição da referência para o Servidor de Nomes, consulta de um objeto distribuído através da invocação do método *resolve()* do Servidor de Nomes e invocação de operações remotas disponibilizadas pelo objeto distribuído. Uma das vantagens apresentadas pelo CCM é que clientes CORBA 2.x continuam utilizando servidores CCM, desde que as interfaces do serviço sejam preservadas. Esses clientes são denominados *component-unaware clientes*, em contrapartida com os *component-aware clientes*, que exploram recursos do CCM tais como obtenção e navegação entre *ports*.

A figura 3.15 apresenta os passos necessários para a criação de uma nova instância de objeto no CORBA 2.x. Todo *servant* (instância) CORBA 2.x necessita ser ativado em um Adaptador de Objetos, antes do seu uso. As políticas definidas no Adaptador de Objetos, no qual o *servant* foi ativado, definem o comportamento dessa instância em relação à persistência, mapeamento *servant*-objetos, dentre outras funcionalidades. O primeiro passo para criação de uma nova instância CORBA 2.x consiste na obtenção de uma referência para o Adaptador de Objetos padrão (raiz) do ORB (passo 1). De posse dessa referência, novos adaptadores podem ser criados (passo 2) com o objetivo de definir políticas apropriadas para o *servant* em questão (passo 3). Após a configuração do adaptador, uma nova instância do *servant* é criada (passo 4) e ativada no adaptador que contém as políticas adequadas para aquele *servant* (passo 5).

A abordagem CORBA 2.x para criação e remoção de novas instâncias de componentes, apesar de constituir a abordagem mais flexível e poderosa para configuração das políticas do Adaptador de Objetos,

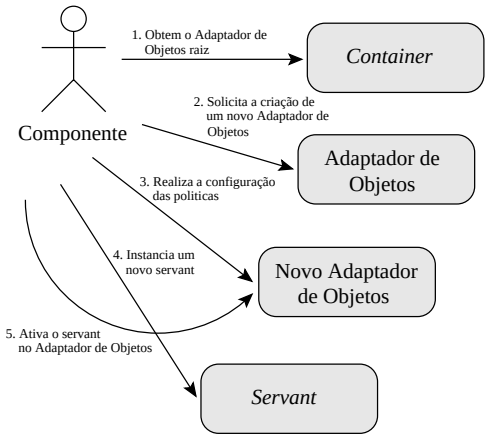


Figura 3.15: Criação de uma nova instância de objeto no CORBA 2.x

requer um trabalho considerável por parte do desenvolvedor. Além disso, o desenvolvimento dessas operações requer experiência e conhecimento especializado sobre a arquitetura CORBA, estando mais sujeito a erros quando realizado por desenvolvedores iniciantes.

Devido ao fato de ser a única alternativa disponível nas primeiras versões do CIAO, esta foi a primeira solução utilizada para a criação e remoção dinâmica dos objetos do DAIS. A criação de objetos de sessão, grupos e *iterators* era realizada pela implementação dos passos descritos acima, consistindo uma solução complexa e que exigia alterações em código sempre que alguma mudança nas políticas destes objetos era necessária.

3.3.2.2 2ª solução: utilização de componentes home

O CCM especifica que todo componente implementado pelo desenvolvedor possui um componente *home* associado. Esse componente *home* disponibiliza operações para criação de novas instâncias (métodos *factory*) e para obtenção de instâncias já criadas (métodos *finder*). Todas as operações descritas anteriormente para a configurações de adaptadores, ativação do *servant* e registro no Servidor de Nomes são realizadas automaticamente pelo *container* de componentes, poupando o desenvolvedor da implementação deste código.

Com essa abordagem, os passos necessários para a criação de uma nova instância de componente são:

1. Aquisição da referência para o componente *home* do componente em questão: durante a implantação de um componente, o *home* correspondente a este componente pode ser registrado no Servidor de Nomes, possibilitando a aquisição dessa referência por clientes interessados na criação de novas instâncias.

2. Invocação do método *factory* para criação da nova instância: uma vez obtida a referência para o

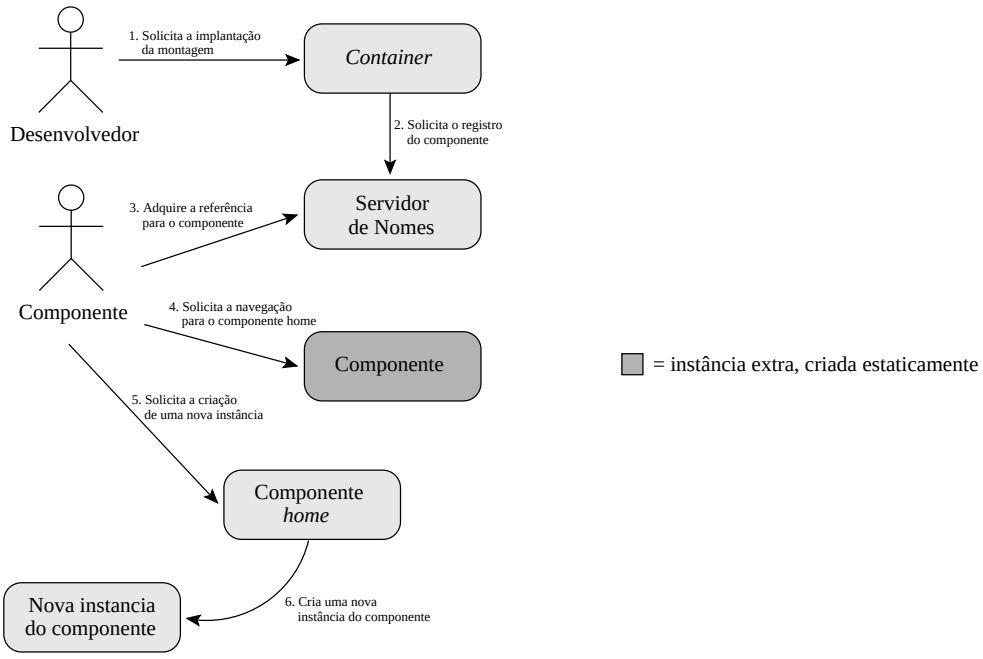


Figura 3.16: Criação de uma nova instância de componente a partir de componentes *home*

componente *home*, novas instâncias podem ser criadas através de invocações aos métodos *factory*, disponibilizados pelo componente *home*.

Com a utilização dos componentes *home*, o desenvolvedor tem à disposição um mecanismo, de fato orientado a componentes, para a criação de novas instâncias. Todas as operações subjacentes para a criação desta instância são realizadas pelo *container* e as políticas do componente são facilmente ajustadas através de redefinições feitas no descritor XML de implantação ou no arquivo CIDL correspondente àquele componente.

A figura 3.16 apresenta os passos necessários para a criação de uma nova instância utilizando os componentes *home* do CCM. Após realizada a implantação (passo 1), uma primeira instância do componente em questão deve ser criada e registrada (passo 2) no Servidor de Nomes (operações realizadas pelo DAnCE). Para a criação de uma nova instância, o Servidor de Nomes deve ser consultado com o objetivo de adquirir uma referência para a instância já criada (passo 3). Utilizando as API's de navegação do CCM, uma referência para o componente *home* é obtida através da referência para o componente já instanciado (passo 4). De posse do componente *home*, solicita-se a criação de uma nova instância a partir da invocação dos métodos do tipo *factory* (passo 5). O componente *home* cria a nova instância e retorna esta nova referência para a entidade solicitante (passo 6).

Apesar dos benefícios citados, essa abordagem baseada em componentes *home* apresenta algumas desvantagens. O CCM define operações, a serem implementadas por todo componente, que deverão ser oportunamente invocadas de modo a gerenciar o ciclo de vida deste componente. Essas operações,

denominadas métodos de *callback*, são: pré-ativação, ativação, pós-ativação, passivação e remoção. Cada método de *callback* representa uma mudança de estado no ciclo de vida do componente e o desenvolvedor pode escrever código a ser executado nestas mudanças. Quando uma instância é criada a partir do seu componente *home*, a invocação dos métodos correspondentes às mudanças de estado fica a cargo do desenvolvedor. O DAnCE, apresentado na sub-seção 2.5.1.1, é o mecanismo disponibilizado pelo CIAO para gerenciamento automático da invocação desses métodos, porém ele não é utilizado quando o componente é manualmente instanciado a partir do *home* correspondente. Para que o DAnCE seja utilizado, o componente deve ter sido criado diretamente pelo XML descritor de implantação ou através do uso do ReDaC, apresentado na próxima sub-seção.

Em particular, na implementação atual do CIAO, os componentes *home* só são criados no momento da criação, via descritor XML, da primeira instância do componente. Além disso, enquanto essa primeira instância é registrada no Servidor de Nomes, o componente *home* associado não é registrado. É necessário que uma instância seja criada estaticamente somente para tornar possível a navegação, desta instância, para o componente *home* associado. Essa primeira instância não é utilizada efetivamente para fins de negócio.

A utilização de componentes *home* foi a segunda solução utilizada, na implementação do ARCOS, para a criação dinâmica de novas instâncias de componentes. As desvantagens apresentadas acima tornavam essa solução ainda trabalhosa para o desenvolvedor, além de requerer a utilização de uma instância extra, somente para aquisição do componente *home*.

3.3.2.3 3ª solução (atual): utilização do ReDaC

O ReDaC, conforme apresentado na sub-seção 2.5.1.2, é o mecanismo disponibilizado pelo CIAO para reimplantação e reconfiguração de montagens. Uma montagem (*assembly*) define quais componentes serão instanciados na aplicação, os valores dos atributos definidos por tais componentes e como estes se conectam com outros componentes de modo a prover a funcionalidade requerida. Essas montagens são especificadas através de arquivos XML descritores de implantação, escritos manualmente pelo desenvolvedor ou gerados automaticamente por alguma ferramenta de montagem (*assembly tool*).

O DAnCE, apresentado na sub-seção 2.5.1.1, realiza a leitura desse arquivo descritor no momento da implantação da montagem e gerencia todas as atividades de criação das instâncias, configuração dos atributos e estabelecimento das conexões. Utilizando somente o DAnCE, alterações na montagem requerem a interrupção do sistema, a alteração do arquivo descritor e a reimplantação da nova montagem.

O ReDaC permite a alteração de montagens já implantadas, sem requerer a interrupção do serviço. A abordagem consiste na utilização de um outro arquivo descritor contendo a nova montagem. Desta forma, se uma determinada instância da nova montagem não estava presente na montagem anterior, esta instância será dinamicamente criada pelo ReDaC. Se uma determinada instância, presente na montagem

anterior, não está representada no novo arquivo descritor, esta instância será dinamicamente removida pelo ReDaC. O mesmo ocorre para conexões adicionadas, retiradas ou reconfiguradas, bem como novos valores para atributos de componentes.

Conforme já apresentado, o ReDaC pode ser utilizado de duas formas: através de execuções manuais do *PlanLauncher* ou via programação. Independente da abordagem utilizada, nota-se que o ReDaC é uma abordagem poderosa para reimplantação e reconfiguração de montagens visto que, através deste mecanismo, é possível realizar qualquer alteração no descritor XML que representa a aplicação. Além disso, todas as invocações dos métodos de *callback* são automaticamente realizadas pelo DAnCE e não é necessário a existência de uma instância extra para aquisição do componente *home*.

A figura 3.17 apresenta os passos necessários para a criação de uma nova instância utilizando o ReDaC, via programação. Inicialmente, o desenvolvedor solicita a implantação da montagem inicial (passo 0). O *ExecutionManager* realiza a implantação (passo 1) e disponibiliza a representação em memória do Plano de Implantação. O componente que deseja fazer uma reimplantação/reconfiguração adquire o Plano de Implantação (passo 2) e solicita manipulações neste plano (passo 3) de modo a refletir a nova montagem (inclusão/exclusão de instâncias e conexões, mudanças nos valores de atributos, etc). Essas manipulações são realizadas por uma classe utilitária, desenvolvida no projeto do ARCOS, denominada *ReDaCUtils*. Os objetivos e métodos disponibilizados por essa classe são apresentados na próxima sub-seção. Após ocorridas as manipulações, o novo Plano de Implantação é enviado de volta ao *ExecutionManager*, que realiza a reimplantação da montagem (passo 4). Esse ciclo se repetirá sempre que houver necessidade de mudanças na configuração da montagem atualmente implantada, sem requerer a interrupção do serviço.

Por todos os benefícios acima citados, o ReDaC é a solução atualmente utilizada no ARCOS para a criação dinâmica de novas instâncias dos componentes que implementam o padrão DAIS (sessões, *group managers*, *iterators* etc). Essa solução é apresentada em detalhes na próxima sub-seção.

3.3.3 ARCOS, DAnCE e ReDaC

O DAnCE e o ReDaC são duas tecnologias que, em conjunto, representam uma boa solução para o desenvolvimento de sistemas flexíveis, escaláveis e com possibilidade de manutenção e evolução sem requerer a interrupção do serviço. O DAnCE e o ReDaC foram amplamente utilizados na implementação do serviço de aquisição de dados do ARCOS (padrão DAIS).

Para a criação dinâmica de novas instâncias de componentes utilizou-se o ReDaC via programação. Nessa abordagem, o arquivo XML descritor da implantação é representado, em memória, como uma estrutura (*struct*) CORBA do tipo *DeploymentPlan*. Conforme apresentado no código 3.9, essa estrutura contém campos, tais como *implementation*, *instance* e *connection* (linhas 5, 6 e 7, respectivamente), que representam os blocos XML presentes no arquivo descritor.

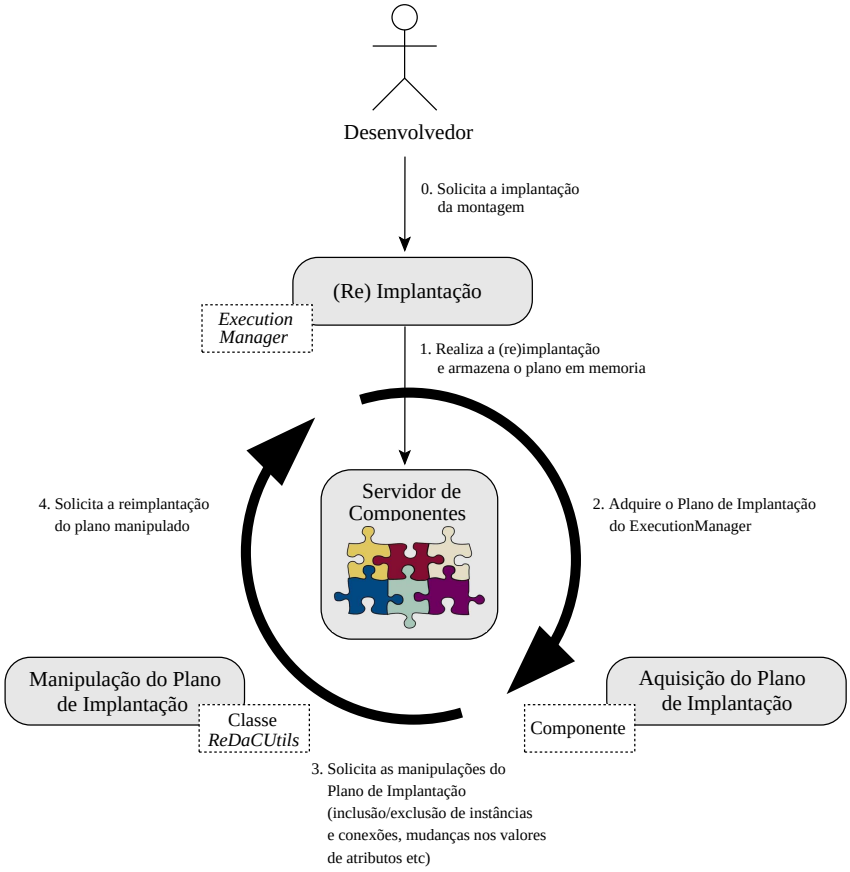


Figura 3.17: Criação de uma nova instância de componente utilizando o ReDaC, via programação

Código 3.9: Estrutura IDL representando o plano de implantação

```
1  struct DeploymentPlan {
2      string label;
3      string UUID;
4      ComponentInterfaceDescription      realizes;
5      MonolithicDeploymentDescriptions   implementation;
6      InstanceDeploymentDescriptions     instance;
7      PlanConnectionDescriptions         connection;
8      PlanPropertyMappings               externalProperty;
9      ImplementationDependencies         dependsOn;
10     ArtifactDeploymentDescriptions     artifact;
11     Properties                         infoProperty;
12 };
```

Os campos dessa estrutura são seqüências[1] CORBA que contêm outras estruturas representando os blocos presentes no arquivo descritor. Por exemplo, o campo *instance* é uma seqüência contendo estruturas do tipo *InstanceDeploymentDescription*, apresentada no código 3.10. Todas informações presentes no descritor XML são representados através de campos dessa estrutura. Os campos *name* e *node* (linhas 2 e 3) representam, respectivamente, o nome dado à instância e o nó onde esta instância será criada. O campo *implementationRef* armazena o índice que representa o tipo dessa instância na seqüência de tipos, definida no código 3.9 pelo campo *implementation*. O campo *configPropery* é uma seqüência que contém, para cada atributo do componente, o valor atribuído a este atributo nesta instância. A linha

[1] Tipos, definidos na linguagem IDL, para representar uma agregação homogênea, como um vetor ou lista encadeada.

11 define a seqüência de estruturas do tipo *InstanceDeploymentDescription*. Seqüências semelhantes são definidas para os demais campos da estrutura *DeploymentPlan*, sendo todas apresentadas no arquivo *Deployment_Data.idl*, presente no CIAO.

Código 3.10: Estrutura IDL representando uma instância do plano de implantação

```
1  struct InstanceDeploymentDescription {
2     string name;
3     string node;
4     ::CORBA::StringSeq source;
5     unsigned long       implementationRef;
6     Properties          configProperty;
7     InstanceResourceDeploymentDescriptions deployedResource;
8     InstanceResourceDeploymentDescriptions deployedSharedResource;
9  };
10
11 typedef sequence < InstanceDeploymentDescription > InstanceDeploymentDescriptions;
```

Desta forma, a manipulação, via programação, do plano de implantação consiste na navegação entre as seqüências e estruturas que definem tipos, instâncias, atributos e conexões realizando a inserção/-remoção de novos elementos nestas seqüências (para a criação/remoção de instâncias ou conexões) ou alterações de valores já existentes (para reconfiguração de atributos de instâncias, por exemplo). A estrutura *DeploymentPlan* é obtida através da invocação do método *getPlan()*, disponibilizado pelo objeto *ExecutionManager* do DAnCE. Após as devidas alterações na estrutura do plano de implantação, o método *perform_redeployment()* do *ExecutionManager* deve ser executado, realizando a reimplantação/-reconfiguração *on-line* da nova montagem.

Em suma, os passos necessários para criação e reimplantação *on-line* de uma nova montagem são:

1. Aquisição do *ExecutionManager*: conforme apresentado na sub-seção 2.5.1.1, o *ExecutionManager* é o objeto responsável pelo gerenciamento do processo de implantação em um ou mais domínios. Esse objeto armazena o plano de implantação e disponibiliza o método *getPlan()*, para obtenção deste plano. Para manipulação do plano de implantação via código é necessário consultar o Servidor de Nomes para obter a referência do objeto *ExecutionManager*. Ressalta-se que para que o *ExecutionManager* seja automaticamente registrado no Servidor de Nomes ele deve ser executado com o parâmetro '-n'.

2. Obtenção do plano de implantação: com a invocação do método *getPlan()*, disponibilizado pelo *ExecutionManager*, o plano de implantação é retornado, sob a forma de uma estrutura do tipo *DeploymentPlan*.

3. Manipulação do plano de implantação: navegação entre as seqüências do plano de implantação realizando as devidas alterações, tais como inclusão/remoção de instâncias e conexões e reconfiguração de atributos.

4. Reimplantação da montagem alterada: após as devidas alterações no plano de implantação, este deve ser reimplantado através da invocação do método *perform_redeployment()*, disponibilizado pelo *ExecutionManager*.

Vale ressaltar que, por questões de concorrência, o plano de execução deve ser obtido do *ExecutionManager* no último instante possível e garantias devem ser implementadas para que se realize exclusão mútua na região compreendida entre a aquisição do plano de implantação e a sua efetiva reimplantação.

3.3.3.1 A classe ReDaCUtils

A manipulação direta e rotineira das estruturas que definem o plano de implantação é uma atividade enfadonha e propensa a erros. A navegação até o campo a ser alterado, o incremento/decremento do tamanho da seqüência e a inclusão/exclusão da nova informação são atividades freqüentemente realizadas no ARCOS.

Com o objetivo de localizar os códigos para alteração do plano de implantação e de prover uma interface mais amigável para manipulação deste plano, o ARCOS implementa uma classe utilitária que provê métodos que facilitam a manipulação do plano de implantação. Essa classe, denominada *ReDaCUtils*, é apresentada no código 3.11.

Código 3.11: Classe *ReDaCUtils* para manipulação do plano de implantação

```
1  class ARCOSUTILS_Export ReDaCUtils
2  {
3  public:
4      static void add_instance (::Deployment::DeploymentPlan_var &deployment_plan,
5                                const char *instance_name, const char *node_name,
6                                const char *type, const char *ns_name)
7                               ACE_THROW_SPEC ((ImplementationNotFound));
8
9      static void remove_instance (::Deployment::DeploymentPlan_var &deployment_plan,
10                                  const char *instance_name)
11                                  ACE_THROW_SPEC ((InstanceNotFound));
12
13     static void add_connection (::Deployment::DeploymentPlan_var &deployment_plan,
14                                 const char *connection_name, const char *port_name,
15                                 const char *facetsource_instance,
16                                 const char *receptaclesink_instance,
17                                 ConnectionType ct)
18                                 ACE_THROW_SPEC ((InstanceNotFound));
19
20     static void remove_connection (::Deployment::DeploymentPlan_var &deployment_plan,
21                                    const char *connection_name)
22                                    ACE_THROW_SPEC ((ConnectionNotFound));
23
24     static void show_all_instances (::Deployment::DeploymentPlan_var &deployment_plan);
25
26     static void show_all_connections (::Deployment::DeploymentPlan_var &deployment_plan);
27 };
28 #endif /* ARCOSREDACUTILS_H */
```

Os métodos disponibilizados pelo class ReDaCUtils são:

- *add_instance()* (linhas 4 a 7): adiciona uma nova instância de componente no plano de implantação. Os parâmetros recebidos pelo método são: o plano de implantação adquirido do *ExecutionManager* (*deployment_plan*), o nome da instância (*instance_name*)[2], o nome do nó no qual será criada a instância (*node_name*)[3], o tipo da instância (*type*)[4] e o nome que será utilizado no registro dessa nova instância no Servidor de Nomes (*ns_name*). Caso o tipo da instância não seja encontrada, em nenhum bloco do tipo *implementation*, a exceção *ImplementationNotFound* é gerada.

- *remove_instance()* (linhas 9 a 11): remove uma instância presente no plano de implantação. Os parâmetros recebidos pelo método são: o plano de implantação (*deployment_plan*) e o nome da instância a ser removida (*instance_name*). O nome da instância deve corresponder à *tag name* de algum bloco do tipo *instance*. Caso este bloco não exista, a exceção *InstanceNotFound* é gerada.

- *add_connection()* (linhas 13 a 17): adiciona uma nova conexão (via faceta/receptáculo ou produtor/depósito de eventos) entre duas instâncias de componentes já existentes. Os parâmetros recebidos pelo método são: o plano de execução (*deployment_plan*), o nome de conexão (*connection_name*)[5], o nome do *port* a ser conectado (*port_name*)[6], o nome da instância que contém a faceta/produtor (*facetsource_instance*), o nome da instância que contém o receptáculo/depósito (*receptaclesink_instance*) e o tipo da conexão a ser criada (faceta-receptáculo ou produtor-depósito). Os nomes dessas instâncias devem coincidir com a *tag name* de algum bloco do tipo *instance*. Caso alguma dessas duas instâncias não seja encontrada, a exceção *InstanceNotFound* é gerada.

- *remove_connection()* (linhas 19 a 21): remove uma conexão já estabelecida entre duas instâncias do plano de implantação. Os parâmetros recebidos pelo método são: o plano de implantação (*deployment_plan*) e o nome da conexão a ser removida (*connection_name*). Caso o nome da conexão não seja encontrada, a exceção *ConnectionNotFound* é gerada.

- *show_all_instances()* (linha 23): apresenta na tela todas as informações sobre todas as instâncias presentes no plano de implantação. Esse método recebe como parâmetro o plano de implantação em questão (*deployment_plan*) e é utilizado para fins de depuração.

- *show_all_connections()* (linha 25): apresenta na tela todas as informações sobre todas as conexões presentes no plano de implantação. Esse método recebe como parâmetro o plano de implantação em questão (*deployment_plan*) e é utilizado para fins de depuração.

[2] Nome que será utilizado no atributo *id* da *tag instance* e na tag *name* do bloco *instance*, presentes no descritor XML de implantação.
[3] Este nó deverá estar descrito no mapa de nós do domínio de implantação, apresentado na seção 2.5.1.1.
[4] Este tipo será utilizado na tag *implementation* do bloco *instance* e deve corresponder ao valor da *tag name* de algum bloco do tipo *implementation*.
[5] Nome que será utilizado na *tag name* do bloco *connection*, presente no descritor XML de implantação.
[6] Deve coincidir com o nome do *port* descrito no arquivo IDL do componente. Assume-se que esse nome será o mesmo tanto na faceta/produtor quanto no receptáculo/depósito.

A classe *ReDaCUtils* foi extensivamente utilizada no ARCOS para a criação dinâmica dos componentes que implementam o padrão DAIS. A criação de sessões de acesso a dados, grupos de aquisição e *iterators* é realizada através de invocações dos métodos da classe *ReDaCUtils*, criando novas instâncias, realizando as conexões apresentadas na tabela 3.1 e retornando referências destes recém-criados componentes para o cliente DAIS em questão.

Futuramente, a classe *ReDaCUtils* pode ser estendida de modo a disponibilizar métodos para a manipulação de outras informações do plano de implantação, bem como para a geração automática de um novo arquivo descritor XML que reflita o estado atual do plano de implantação.

3.3.4 MPC (Make Project Creator)

O desenvolvimento de sistemas distribuídos baseados no CIAO é geralmente caracterizado pela presença de uma quantidade considerável de arquivos contendo definições IDL e CIDL, código-fonte, descritores de implantação, dentre outras informações. A utilização de ferramentas, geralmente denominadas *"build tools"*, que auxiliem o processo de compilação é um fator importante para a gerência segura do processo de desenvolvimento.

Uma ferramenta bastante conhecida para gerência do processo de compilação de sistemas escritos em C/C++ é o *make* [26]. O *make* faz uso de arquivos de configuração, denominados *makefiles*, para gerenciar o processo de compilação de sistemas grandes. Dentre as suas vantagens, pode-se destacar a compilação facilitada de todo o projeto e a compilação apenas daqueles arquivos que foram alterados. Entretanto, a construção dos arquivos *makefiles* é uma tarefa enfadonha e que requer conhecimento especializado. A utilização de ferramentas, tais como o *automake* [25], reduzem consideravelmente o esforço para geração desses arquivos, porém não facilitam o processo de geração de *makefiles* voltados para o ACE, TAO e CIAO.

O MPC (*Make Project Creator*) [66] é uma ferramenta para geração de *makefiles* voltados especificamente para a compilação de sistemas baseados no ACE, TAO e CIAO. Com o MPC, os arquivos *makefiles* são gerados a partir de arquivos de configuração mais simples e que podem facilmente ser construídos pelo próprio desenvolvedor. O apêndice B apresenta um exemplo de arquivo de configuração do MPC, explicando detalhadamente o seu funcionamento. O MPC foi extensivamente utilizado na implementação do ARCOS, possibilitando que este possa ser compilado através de uma única execução do comando *make*.

3.3.5 Implementação dos módulos

Nesta sub-seção são apresentados aspectos técnicos sobre a implementação dos módulos de aquisição de dados, controle e supervisão do ARCOS. No módulo de aquisição de dados é apresentado como o

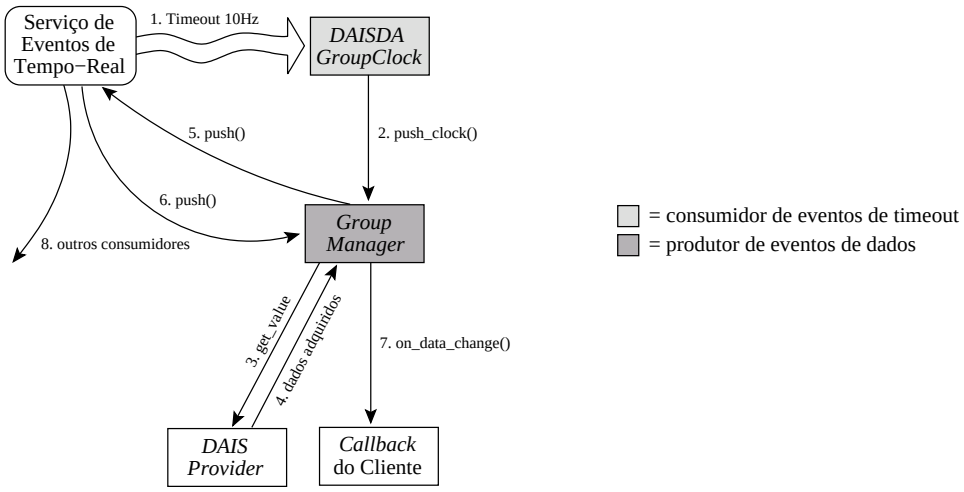


Figura 3.18: Implementação do *loop* de aquisição através do uso dos eventos de *timeout*

Serviço de Eventos de Tempo-Real do TAO foi utilizado para sincronizar a entrega de itens de grupos de aquisição e como o componente *DAISDAGroupManager* realiza, ao mesmo tempo, os papéis de produtor e consumidor de eventos. No módulo de controle é apresentado como este módulo interage com o módulo de aquisição de dados para o fechamento da malha. No módulo de supervisão é apresentada a arquitetura utilizado do lado do cliente para possibilitar a comunicação com os componentes disponibilizados pelo ARCOS.

3.3.5.1 Aquisição de dados

Conforme apresentado na sub-seção 3.2.1, o módulo de aquisição de dados do ARCOS é formado pela implementação do padrão DAIS. Para cada componente da figura 3.4 foi desenvolvido um executor que implementa as operações padronizadas pelas interfaces do DAIS. Os componentes DAIS criados sob demanda, tais como sessões de acesso a dados, grupos de aquisição e *iterators* são instanciados através do mecanismo baseado no ReDaC, apresentado na sub-seção 3.3.3.

Após a criação do grupo e inclusão das folhas DAIS que representam os dados a serem adquiridos, o cliente DAIS deve, periodicamente e na freqüência informada, ter o método *on_data_change()* do seu objeto de *callback* invocado pelo servidor DAIS.

Conforme apresentado na seção 2.4.1.2, uma das extensões proporcionadas pelo Serviço de Eventos de Tempo-Real do TAO é o suporte a eventos de *timeout*. Esses eventos se comportam da mesma forma que eventos gerados por produtores, porém são gerados automaticamente pelo canal de eventos, na freqüência informada pelo desenvolvedor. Esses eventos de *timeout* foram utilizados na implementação do componente *DAISDAGroupClock*, apresentado nas figuras 3.4 e 3.18.

Para cada grupo de aquisição criado, são criadas instâncias dos componentes *DAISDAGroupManager* e *DAISDAGroupClock*, responsáveis pelo controle daquele grupo. O componente *DAISDAGroupClock* é

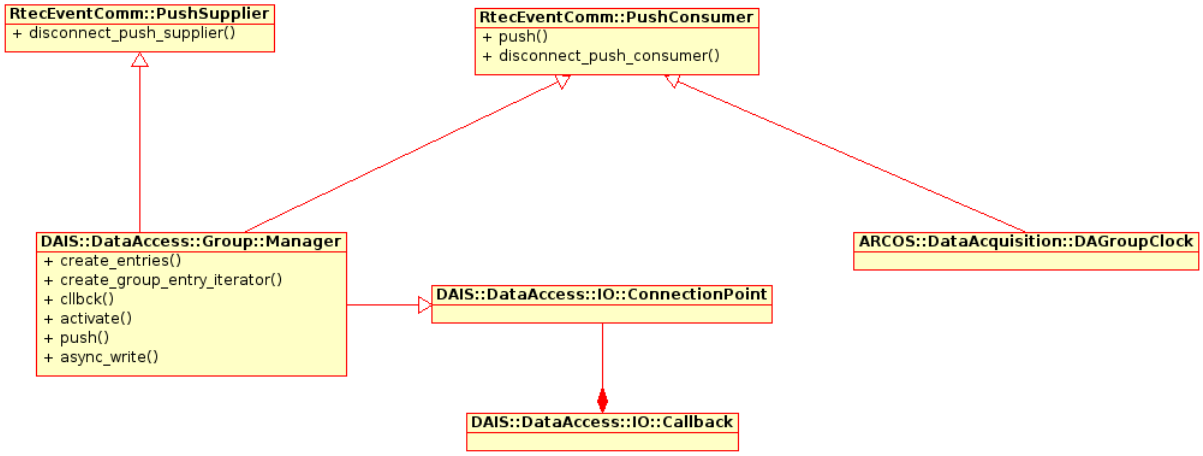


Figura 3.19: Implementação dos componentes *DAISDAGroupManager* e *DAISDAGroupClock*

registrado, no Serviço de Eventos de Tempo-Real, como um consumidor de eventos de *timeout*, recebendo eventos na freqüência de aquisição informada para o grupo (passo 1). Para cada evento de *timeout* recebido, o componente *DAISDAGroupClock* informa, via conexão entre produtor e depósitos de eventos CCM, que o componente *DAISDAGroupManager* precisa adquirir novos dados dos dispositivos (passo 2). Para isso, o *DAIS Provider* devidamente conectado é consultado (passos 3 e 4) e os dados adquiridos são enviados para o Serviço de Eventos de Tempo-Real (passo 5). Esses dados produzidos serão recebidos por todos os consumidores interessados nesse evento (passo 8), inclusive o próprio componente *DAISDAGroupManager* (passo 6). Somente então, o *DAISDAGroupManager* atualiza o cliente através da invocação do método *on_ data_ change()* do *callback* (passo 7). Evitou-se o envio, dos dados de aquisição, de forma direta entre o *DAISDAGroupManager* e o *callback* do cliente, devido ao fato de o Serviço de Eventos de Tempo-Real consistir um ponto de escalonamento, realizando o *dispatch* de eventos baseado em prioridades definidas pelo Serviço de Escalonamento.

A figura 3.19 apresenta o diagrama de classes referentes à implementação do *DAISDAGroupManager* e do *DAISDAGroupClock*. As classes *RtecEventComm::PushSupplier* e *RtecEventComm::PushConsumer*, disponibilizados pelo Serviço de Eventos de Tempo-Real do TAO, são as classes utilizadas para a implementação de produtores e consumidores, respectivamente. Os consumidores, em particular, devem implementar o método *push()*, invocado pelo Serviço de Eventos quando uma nova mensagem deve ser entregue. Esse é o método invocado periodicamente pelo Serviço de Eventos, para a entrega dos eventos de *timeout*.

Conforme apresentado na figura 3.19, o componente *DAISDAGroupClock* é um consumidor de eventos e o método *push()* desse componente é invocado periodicamente, pelo Serviço de Eventos, para a entrega dos eventos de *timeout*. O componente *DAISDAGroupManager*, por sua vez, se comporta como um produtor e consumidor de eventos de atualização dos dados da planta.

3.3.5.2 Controle

O módulo de controle é formado por um conjunto de componentes, propostos pelo ARCOS, para implementação de malhas fechadas de controle. Conforme apresentado na sub-seção 3.2.2, os principais componentes desse módulo são: o *ControlManager*, o *ControlManagerDAISFacade* e os *Controllers* específicos.

A execução do *loop* de controle é dirigida pela recepção, no componente *ControlManagerDAISCallback*, dos dados que representam a leitura da variável controlada. A cada informação recebida, o componente *ControlManager* é notificado e o controlador devidamente conectado é invocado para realizar as operações de controle.

Para facilitar as atividades de comunicação com o servidor DAIS, uma classe que implementa o padrão de projeto *facade* foi criada com o objetivo de agrupar, em um único método, atividades que iriam requerer algumas invocações a métodos do servidor DAIS. Desta forma, os clientes utilizam uma API com um maior nível de abstração e não se preocupam com detalhes específicos do padrão DAIS. Essa classe, denominada *DAISFacade*, é utilizada por qualquer participante do ARCOS que seja um cliente do serviço DAIS. Tanto o componente *ControlManager*, quanto as aplicações de supervisão, fazem uso dessa solução. Detalhes sobre a classe *DAISFacade*, bem como sobre a implementação das aplicações de supervisão são apresentadas na próxima sub-seção.

3.3.5.3 Supervisão

Conforme apresentado na sub-seção 3.2.3, as aplicações com funcionalidades para supervisão (*DAIS Server Browser* e *DAIS Server Manager*) estão reunidas na ferramentas denominada *ARCOS Management Tool* (AMT). O *DAIS Server Browser* é um cliente DAIS genérico e pode ser utilizado para adquirir dados de qualquer implementação de um servidor DAIS, em conformidade com o padrão. O *DAIS Server Manager* monitora as sessões criadas em um servidor e os grupos de aquisição criados em cada sessão.

Conforme ilustrado na figura 3.20, a classe *DAISFacade* reúne um conjunto de métodos que facilita a implementação de clientes DAIS. Além de disponibilizar métodos em um nível de abstração mais alto, essa classe pode ser reutilizada em vários clientes DAIS (tais como o *DAIS Server Browser* e o componente *ControlManager*), constituindo um ponto único de manutenção dessas funcionalidades.

O código 3.12 apresenta os métodos disponibilizados pela classe *DAISFacade*, a saber:

- Construtor 1 (linhas 4 a 6): cria e inicializa o ORB a partir da *string* informada como parâmetro. Todas as opções de inicialização definidas para o CORBA (tais como definição de *endpoint* para o serviço ou localização do Servidor de Nomes) podem ser informados nesse método.
- Construtor 2 (linhas 8 a 10): utiliza uma configuração de ORB e de Servidor de Nomes previamente

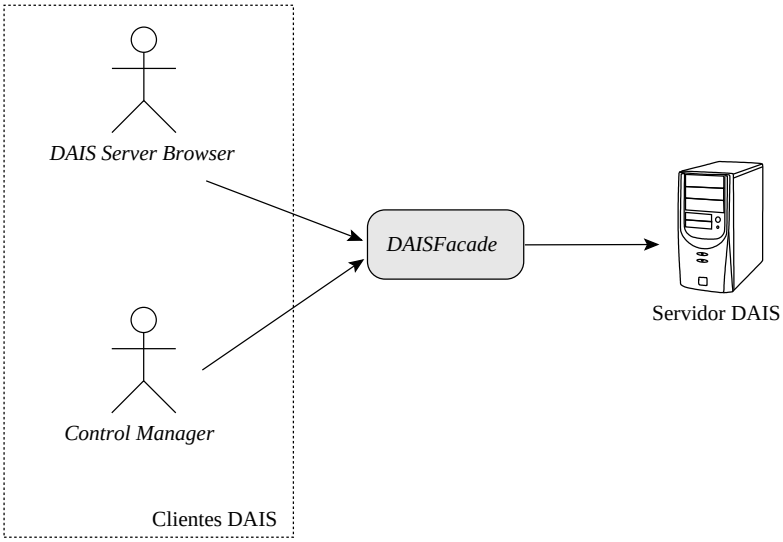


Figura 3.20: Uso da classe *DAISFacade* para implementação de clientes DAIS

criados. Caso esse servidor não seja encontrado no Servidor de Nomes, a exceção *DAISServerNameNotFound* é gerada.

- *init()* (linhas 14 a 17): inicia a conexão com o servidor DAIS registrado, no Servidor de Nomes, com o nome informado como parâmetro.

- *create_data_access_session()* (linhas 19 a 22): cria uma sessão DAIS com o nome informado como parâmetro. A classe *DAISFacade* assume que somente uma sessão DAIS será criada, sendo esta armazenada nas estruturas interna da classe. Caso seja necessário utilizar mais de uma sessão pode-se utilizar uma outra instância da classe *DAISFacade*. Se já existir, no servidor DAIS, uma sessão com o mesmo nome, a exceção *DASessionAlreadyExists* será gerada.

- *get_root()* (linha 24): retorna informações sobre o nó raiz da árvore DAIS disponibilizada pelo servidor. Essa informação é utilizado pelos clientes para a obtenção recursiva da árvore DAIS.

- *find()* (linhas 26 a 28): obtém informações sobre o nó identificado pelo ID informado como parâmetro.

- *find_by_parent()* (linhas 30 e 31): obtém todos os nós filhos do nó informado como parâmetro. Esse método, em conjunto com o método *get_root()*, é utilizado para implementar a obtenção recursiva da estrutura da árvore DAIS disponibilizada pelo servidor.

- *create_data_access_group()* (linhas 33 a 36): método que cria um grupo de aquisição com o nome e taxa de aquisição informados como parâmetros. Esse método retorna uma referência remota para o componente *DAISDAGroupManager* responsável pelo gerenciamento do grupo recém-criado.

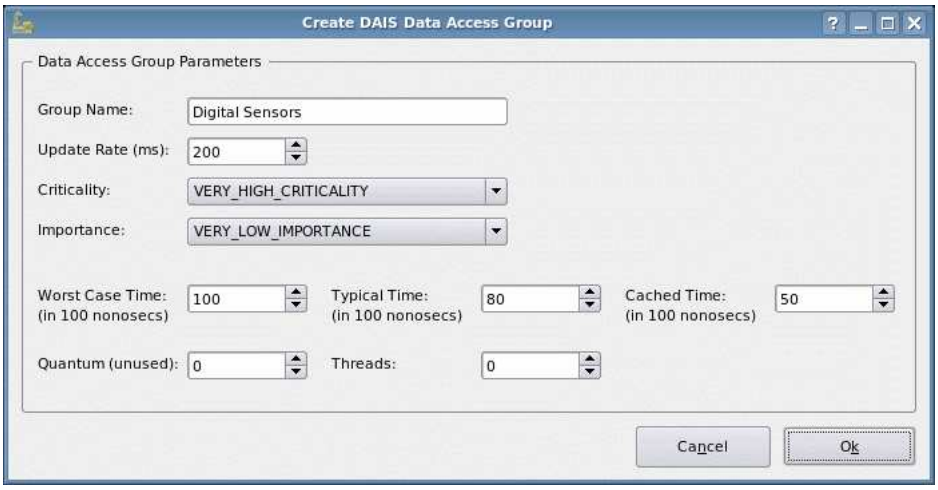


Figura 3.21: Especificação dos parâmetros temporais de um grupo DAIS

- *add_leaf_to_group()* (linhas 38 a 41): método que adiciona uma folha DAIS em um grupo de aquisição. O grupo e a folha DAIS em questão são especificados pelos parâmetros *group* e *id*, respectivamente.

- *add_branch_to_group()* (linhas 43 a 47): método que adiciona, em um grupo de aquisição, todas as folhas de um ramo da árvore DAIS. O grupo e as folhas DAIS em questão são especificados pelo parâmetros *group* e *descrips*, respectivamente.

- *get_group_entries()* (linhas 49 a 51): método que retorna todas as folhas DAIS presentes no grupo especificado pelo parâmetro *group*.

- *remove_entries_from_group()* (linhas 53 a 56): método que remove folhas DAIS de um grupo de aquisição. O grupo e as folhas DAIS em questão são especificados pelo parâmetros *group* e *server_handles*, respectivamente.

Conforme apresentado anteriormente, cada grupo de aquisição de DAIS se comporta, ao mesmo tempo, como um produtor e consumidor do Serviço de Eventos de Tempo-Real do TAO. Para cada consumidor e produtor, os parâmetros temporais devem ser informados no momento da conexão, conforme apresentado no código 2.13. O *DAIS Server Browser* solicita, do usuário, a informação desses parâmetros no momento da criação de um grupo de aquisição, conforme apresentado na figura 3.21.

3.4 ARCOS e mecanismos básicos para tempo-real

Um sistema de tempo-real é aquele onde a sua execução é previsível no domínio lógico (*correctness*) e temporal (*timeliness*) [55, 49, 80]. Para que essa previsibilidade seja garantida, é necessário a existência

Código 3.12: Classe *DAISFacade* para implementação de clientes DAIS

```
class DAISFacade
{
public:
    DAISFacade(std::string orb_parameters)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::NameServiceNotFound,
                     ::ARCOS::DAISExceptions::UnknownException));

    DAISFacade(CORBA::ORB_var orb, CosNaming::NamingContext_var root_nc)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::NameServiceNotFound,
                     ::ARCOS::DAISExceptions::UnknownException));

    virtual ~DAISFacade();

    void init (const char * dais_server_name)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::DAISServerNameNotFound,
                     ::ARCOS::DAISExceptions::InvalidDAISServerName,
                     ::ARCOS::DAISExceptions::UnknownException));

    void create_data_access_session(const char *session_name)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::DASessionAlreadyExists,
                     ::ARCOS::DAISExceptions::DASessionAlreadyCreated,
                     ::ARCOS::DAISExceptions::UnknownException));

    ::DAIS::Node::Description_var get_root();

    ::DAIS::Node::Description * find (const ::DAIS::ResourceID & node)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::ResourceIDNotFound,
                     ::ARCOS::DAISExceptions::UnknownException));

    ::std::vector< ::DAIS::Node::Description > find_by_parent (::DAIS::ResourceID parent)
    ACE_THROW_SPEC ((::DAIS::Node::IHome::UnknownResourceID));

    ::DAIS::DataAccess::Group::Manager_var create_data_access_group(
                    const char *group_name, unsigned long update_rate)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::DAGroupAlreadyExists,
                     ::ARCOS::DAISExceptions::UnknownException));

    ::DAIS::DataAccess::GroupEntry::Results * add_leaf_to_group (
                     ::DAIS::DataAccess::Group::Manager_ptr group,
                     ::DAIS::ResourceID id, ::DAIS::ItemErrors_var errors)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::UnknownException));

    ::DAIS::DataAccess::GroupEntry::Results * add_branch_to_group (
                     ::DAIS::DataAccess::Group::Manager_ptr group,
                     ::DAIS::DataAccess::GroupEntry::Descriptions descrips,
                     ::DAIS::ItemErrors_var errors)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::UnknownException));

    ::std::vector< DAIS::DataAccess::GroupEntry::DetailedDescription > get_group_entries (
                     ::DAIS::DataAccess::Group::Manager_ptr group)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::UnknownException));Control

    ::DAIS::ItemErrors * remove_entries_from_group (
                     ::DAIS::DataAccess::Group::Manager_ptr group,
                    const ::DAIS::ServerItemHandles & server_handles)
    ACE_THROW_SPEC ((::ARCOS::DAISExceptions::UnknownException));
};
```

de um mecanismo básico para tempo-real que envolve o uso de canais previsíveis de comunicação, sistemas operacionais de tempo-real, linguagens de programação e soluções de *middleware* voltados para tempo-real. O ARCOS, devido ao fato de utilizar o CIAO, TAO e ACE como tecnologias subjacentes, representa um esforço para viabilizar o desenvolvimento de aplicações com restrições temporais. Nesta seção será apresentado como o ARCOS se relaciona com esses mecanismos básicos.

Atualmente, o TAO disponibiliza o Serviço de Eventos de Tempo-Real como um mecanismo previsível para comunicação do tipo muitos-para-muitos, desacoplado e assíncrono (não-bloqueante). Conforme apresentado na sub-seção 2.4.1.2, o Serviço de Eventos solicita ao Serviço de Escalonamento a prioridade e a sub-prioridade relacionada a uma determinada mensagem. Essas prioridades são utilizadas, pelo módulo de *dispatch* do Canal de Eventos, para indicar a fila (e a posição nesta fila) na qual será inserida a mensagem a ser entregue. Cada fila de mensagens está associada a uma *thread* do sistema operacional, configurada para executar com uma determinada prioridade. O Serviço de Eventos, e conseqüentemente seu módulo de *dispatch*, executam como uma tarefa de alta prioridade, compondo um cenário freqüentemente reconhecimento como **escalonamento em dois níveis**. Nesse cenário, uma tarefa de alta prioridade é freqüentemente posta em execução pelo escalonador convencional do sistema operacional. Essa tarefa, por sua vez, realiza o escalonamento das demais atividades, a partir de prioridades definidas. A abordagem do escalonamento em dois níveis é utilizada no *middleware* OSA+ aqui apresentado, e foi a estratégia utilizada no projeto do Serviço de Eventos de Tempo-Real do TAO. Ressalta-se que, para aplicações em sistemas do tipo *hard real-time*, o escalonamento deve ser realizado diretamente pelo sistema operacional, aumentando a previsibilidade e evitando a ocorrência de problemas clássicos como, por exemplo, a inversão de prioridades.

O CIAO define outros pontos de previsibilidade na sua arquitetura como, por exemplo, o *container* de tempo-real. Nesse *container*, componentes com restrições temporais serão hospedados e terão sua execução realizada de forma previsível pelo *middleware*. No momento do projeto e implementação do ARCOS, o *container* de tempo-real não estava disponível de forma estável. Vale ainda ressaltar que, atualmente, o CIAO trabalha somente com políticas *off-line* para análise da escalonabilidade e escalonamento.

Uma tarefa importante a ser realizada por plataformas para sistemas de tempo-real é a execução de testes de escalonabilidade [55, 80, 49]. Esses testes verificam a factibilidade do escalonamento de um conjunto de tarefas sujeitas a restrições temporais. O Serviço de Escalonamento, disponibilizado pelo TAO, realiza o teste de escalonabilidade no momento da execução do método *compute_scheduling()*. Esse método verifica a factibilidade do escalonamento a partir das restrições temporais recebidas através da invocação dos métodos *create()* (para criação de uma nova *RT_Info*) e *set()* (para ajuste das restrições em uma determinada *RT_Info*). Caso o escalonamento seja infactível uma exceção CORBA é gerada para a aplicação cliente.

Em sistemas de tempo-real distribuídos, a rede de comunicação utilizada representa também um

ponto de possíveis imprevisibilidades. Para superar esse problema, redes de comunicação de tempo-real (também chamadas de redes industriais) são freqüentemente adotadas. Projetos para integração de requisitos de qualidade de serviço com tecnologias orientadas a objetos, tal como o *Quality Objects* (QuO) [78], representam passos importantes para a obtenção de uma previsibilidade fim-a-fim em plataformas de software para tempo-real.

Espera-se que versões futuras do CIAO promovam um mapeamento mais direto dos serviços disponibilizados pelo *middleware* com os mecanismos básicos oferecidos pelo sistema operacional de tempo-real e pela rede de comunicação em tempo-real, beneficiando as aplicações construídas com base no ARCOS. Vale ressaltar que esses benefícios não implicarão em alterações na implementação do ARCOS e das aplicações nele baseadas, visto que o problema da previsibilidade temporal é tratado como um aspecto ortogonal à solução de engenharia proposta nesta dissertação.

Capítulo 4

Desenvolvimento de Aplicações Baseadas no ARCOS

O desenvolvimento de sistemas distribuídos complexos é geralmente caracterizado pela utilização de bibliotecas, soluções de *middleware* e *frameworks*. Para isso, o desenvolvedor precisa conhecer as funcionalidades, serviços e abstrações disponibilizados por tais soluções. Em particular, a utilização de *frameworks* do tipo caixa branca requer o conhecimento dos *hot-spots* definidos e um estudo sobre o fluxo de controle implementado.

Neste capítulo será apresentado como o ARCOS pode ser utilizado para desenvolver sistemas distribuídos para S&C. Inicialmente, são apresentadas as instruções para instalação do CIAO e do ARCOS e a estrutura de diretórios disponibilizada pela implementação do ARCOS. Em seguida, são apresentados os passos requeridos para a completa instanciação do *framework* em uma situação particular de S&C e como a ferramenta *ARCOS Assembly Tool* facilita o desenvolvimento dessa atividade. São apresentadas duas aplicações exemplo de uso do ARCOS: um sistema para supervisão de um simulador de reator químico, em um cenário de aquisição de dados a partir de um CLP, e um sistema para controle PID de um piloto automático de veículos. Ao final, são apresentados os experimentos de validação da implementação e os trabalhos correlatos.

4.1 Instalando o CIAO e o ARCOS

O CIAO, assim como o ARCOS, são projetos escritos na linguagem de programação Standard C++. É necessário, portanto, que as ferramentas para desenvolvimento em C++ (compiladores, pré-processadores, *linkers*, depuradores e ferramentas utilitárias) estejam previamente instaladas. Devido ao fato de serem baseados em soluções portáveis tais como o ACE e o Qt, o CIAO e o ARCOS podem ser compilados em uma série de plataformas, dentre elas o ambiente GNU/Linux e o Microsoft Windows. O apêndice D apresenta, detalhadamente, os passos necessários para a compilação e instalação do CIAO e do ARCOS.

A figura 4.1 apresenta a estrutura de diretórios presente no pacote de instalação do ARCOS. O dire-

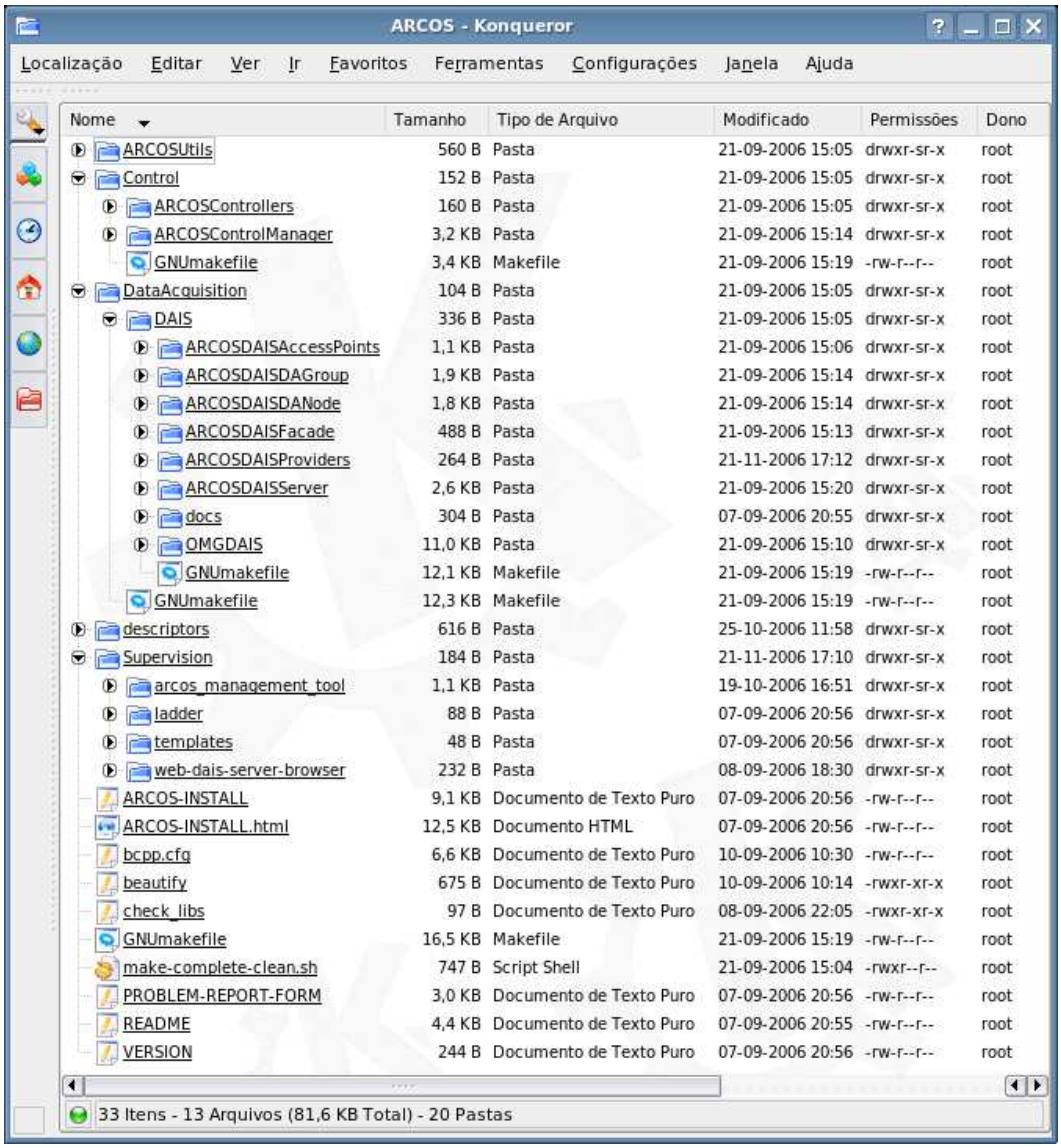


Figura 4.1: Estrutura de diretórios do pacote de instalação do ARCOS

tório raiz do ARCOS é formado pelos sub-diretórios *ARCOSUtils*, *Control*, *DataAcquisition*, *descriptors* e *Supervision*. Os sub-diretórios *DataAcquisition*, *Control* e *Supervision* contêm as implementações dos módulos de aquisição, controle e supervisão, respectivamente. O diretório *ARCOSUtils* contém bibliotecas utilitárias do ARCOS, tais como a classe *ReDaCUtils*, classes para *logging* e para extensões da STL (*Standard Template Library*)[1] [94]. O diretório *descriptors* contém exemplos de arquivos XML descritores de implantação.

O sub-diretório *DataAcquisition* contém o sub-diretório *DAIS*, que armazena a implementação desse padrão. A implementação de outros padrões, tais como o OPC, pode ser realizada e então sugere-se a criação de um novo sub-diretório. O sub-diretório DAIS contém os sub-diretórios *docs*, *OMGDAIS*, *ARCOSDAISServer*, *ARCOSDAISDANode*, *ARCOSDAISDAGroup*, *ARCOSDAISFacade*, *ARCOSDAI-*

[1] A STL é uma biblioteca C++ contendo classes *containers* (estruturas de dados), algoritmos e *iterators*, fortemente parametrizada para uso em uma série de propósitos.

SAccessPoints e *ARCOSDAISProviders*. O sub-diretório *docs* contém documentos sobre o padrão DAIS como, por exemplo, a especificação original disponibilizada pelo OMG. O sub-diretório *OMGDAIS* contém os arquivos IDL padrão do DAIS, bem como os *stubs* e *skeletons* gerados pelo TAO. O sub-diretório *ARCOSDAISServer* contém a implementação dos principais componentes participantes, tais como o *DAISServer* e o *DAISDASession*. Os sub-diretórios *ARCOSDAISDANode* e *ARCOSDAISDAGroup* contêm a implementação dos componentes *DAISDANodeHome* e *DAISDAGroupManager*, respectivamente. O sub-diretório *DAISFacade* contém a implementação do *facade* que simplifica a construção de clientes DAIS. O sub-diretório *ARCOSDAISAccessPoints* contém a definição de interfaces utilizadas por facetas e receptáculos de diversos componentes. O sub-diretório *ARCOSDAISProviders* contém a implementação da interface *IDAISProviderBaseFacet*, utilizada para a construção de novos *DAIS Providers*. Além disso, esse diretório deverá conter todos os *providers* específicos construídos pelo desenvolvedor que utiliza o ARCOS, a exemplo dos *providers* *DAISEthernetPLCProvider* (para aquisição de dados a partir de um CLP conectado em uma rede *Ethernet*) e *DAISSimulatedCarProvider* (para aquisição de dados a partir de um modelo simplificado de veículo automotivo). Esses providers específicos serão apresentados na seção 4.4.

O sub-diretório *Control* é formado pelos sub-diretórios *ARCOSControlManager* e *ARCOSControllers*. O sub-diretório *ARCOSControlManager* contém a implementação dos componentes do módulo de controle do ARCOS, tais como o *ControlManager* e o *ControlManagerDAISCallback*. O sub-diretório *Controllers* contém a implementação da interface *IControllerBaseFacet*, utilizada para a construção de novos controladores. Além disso, esse diretório deverá conter todos os controladores específicos construídos pelo desenvolvedor que utiliza o ARCOS, a exemplo do controlador *PIDController* (implementação do controlador PID tradicional). Esse controlador específico será apresentado na seção 4.4.

O sub-diretório *Supervision* contém as implementações da aplicações clientes disponibilizadas pelo ARCOS: o *ARCOS Management Tool* e a versão web do *DAIS Server Browser*.

4.2 Especializando o ARCOS

O ARCOS é uma solução genérica e reutilizável que permite a construção facilitada de sistemas industriais de S&C. Como todo *framework*, o ARCOS foi projetado e implementado de modo a ser "adaptado" a situações pertencentes a uma determinada classe de sistemas sendo, neste projeto, aquela caracterizada por atividades de aquisição de dados, controle e supervisão, sujeitas à restrições temporais e com requisitos de interoperabilidade. Esta seção apresenta os passos necessários para a construção de uma solução de S&C baseada no ARCOS.

De um modo geral, os passos para a construção de uma nova aplicação de S&C baseada no ARCOS

são:

1. Análise do problema a ser solucionado: essa fase envolve o estudo da situação de S&C a ser solucionada usando o ARCOS. Atualmente, o ARCOS disponibiliza soluções para as atividades de aquisição de dados, controle em malha fechada e supervisão. Outras atividades industriais, tais como gerenciamento de máquinas CNC (Controle Numérico Computadorizado) e programação de CLP's, não são suportadas pelo ARCOS. Nessa fase, deverão ser identificados os pontos onde são realizadas as operações de aquisição de dados e controle. Para cada tecnologia distinta de dispositivo de aquisição e para cada estratégia distinta de controle devem ser implementados os *DAIS Providers* e controladores correspondentes, respectivamente. O ARCOS atualmente suporta a construção de montagens com múltiplos *DAIS Servers* e múltiplos controladores, bastando que cada um deles seja registrado com um nome diferente no Servidor de Nomes (através da tag *RegisterNaming* do descritor de implantação). Cada *DAIS Server* deverá estar conectado a um determinado *DAIS Provider* e cada controlador deverá estar configurado para ler/escrever dados de/em um determinado *DAIS Server*.

2. Implementação do(s) *DAIS Provider(s)*: para cada tecnologia distinta de aquisição de dados, um novo *DAIS Provider* deve ser implementado pelo desenvolvedor, conforme descrito em detalhes na seção 4.2.1.

3. Implementação do(s) Controlador(es): para cada estratégia distinta de controle, um novo controlador deve ser implementado pelo desenvolvedor, conforme descrito em detalhes na seção 4.2.2.

4. Construção do arquivo descritor de implantação: uma vez implementados e compilados o(s) *provider(s)* e o(s) controlador(es), a aplicação deve ser montada a partir da composição dos componentes disponibilizados pelo ARCOS, com os componentes construídos pelo desenvolvedor. Essa composição é realizada pelo arquivo descritor de implantação, conforme descrito em detalhes na seção 4.2.3.

5. Implantação da nova montagem: após criado o arquivo XML descritor da implantação, o sistema pode ser implantado utilizando o DAnCE. Os passos necessários para a implantação de um sistema utilizando o CIAO e o DAnCE foram descritos na seção 2.5.1.1 (código 2.14).

Conforme apresentado na sub-seção 3.2.1, os *hot-spots* do ARCOS são implementados através de heranças realizadas a partir de interfaces definidas pelo ARCOS, caracterizando-o como um *framework* do tipo caixa branca (vide seção 3.2.1.12.3). As implementações dos *hot-spots* de aquisição de dados e controle do ARCOS são apresentadas a seguir.

4.2.1 Implementando um novo DAIS Provider

Os passos requeridos para a implementação de um novo *DAIS Provider* são:

1. Criação do diretório e do arquivo de configuração do MPC do novo *provider*: um novo *DAIS provider* pode ser implementado em qualquer diretório do sistema de arquivos, porém recomenda-se que estes sejam reunidos no diretório *${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/*. O arquivo de configuração do MPC deve ser gerado através do utilitário '*generate_component_mpc .pl <nome do componente>*'. Sugere-se que os nomes dos diretórios sigam o padrão '*ARCOSDAIS<nome do provider>Provider*' como, por exemplo, *ARCOSDAISEthernetPLCProvider* ou *ARCOSDAISParalellPortProvider*. O comando *generate_component_mpc.pl* irá gerar o arquivo '*ARCOSDAIS<nome do provider>Provider.mpc*' e três arquivos requeridos pelo ACE para questões de portabilidade. Edite o arquivo .mpc gerado de modo a refletir as linhas apresentadas no código 4.1. Os objetivos das *tags after*, *libs*, *idlflags*, *cidlflags* e *include* foram apresentados na seção 3.3.4. Conforme será apresentado na seção 4.3, o ARCOS disponibiliza uma ferramenta que automatiza a construção de novas aplicações baseadas no ARCOS: o *ARCOS Assembly Tool*. Com essa ferramenta, a geração desse arquivo .mpc é realizada de forma automática.

2. Criação dos arquivos IDL e CIDL do novo *provider*: após as alterações no arquivo .mpc, o arquivo IDL e CIDL do novo *provider* deverão ser criados pelo desenvolvedor. Um exemplo de arquivo IDL para um novo *provider* foi apresentado na sub-seção 3.2.1.1 e o arquivo CIDL pode ser criado conforme descrito na sub-seção 2.2.1.2. Com a utilização do *ARCOS Assembly Tool*, os arquivos IDL e CIDL são também gerados automaticamente.

3. Geração da versão inicial do executor (vide sub-seção 2.2.1.2) do novo *provider*: o compilador de arquivos CIDL do CIAO é capaz de gerar versões iniciais (com implementações vazias) do executor do componente. Para isso, o compilador (localizado em $CIAO_ROOT/bin/cidlc) deve ser executado com o parâmetro '*–gen-exec-impl*'. Sugere-se que, nesse momento, o desenvolvedor execute os comandos '*mwc.pl*' e '*make*' para verificar a linha de execução do *cidlc* e alterá-la para a inclusão do parâmetro '*–gen-exec-impl*'. Esse parâmetro deve ser inserido antes do trecho '– <nome do componente>'. Esse procedimento gerará os arquivos *<nome do componente_exec.h* e *<nome do componente>_exec.cpp*.

4. Implementação do novo *provider*: os arquivos gerados no passo anterior já contêm implementações vazias dos métodos *build_dais_tree()*, *get_value()* e *set_values()*. Uma atividade importante dessa fase é a definição de como os dados do dispositivo serão organizados, através da construção da árvore DAIS. O desenvolvedor que utiliza o ARCOS deve, portanto, fornecer implementações reais desses

métodos, construindo a árvore DAIS a ser disponibilizada pelo serviço e realizando as operações de comunicação com o dispositivo de aquisição em questão.

5. Compilação do novo *provider*: após a completa implementação do *provider*, a compilação pode ser realizada através dos comandos '*mwc.pl*' e '*make*'.

Código 4.1: Alterações a serem realizadas no arquivo MPC do novo *DAIS Provider*

```
project(ARCOSDAIS<nome do provider>Provider_DnC_stub): ciao_client_dnc {
   after += OMGDAIS_DnC_stub ARCOSDAISDAGroup_DnC_stub \
            ARCOSDAISDANode_DnC_stub ARCOSDAISProviderBase_DnC_stub

   idlflags += -Sc \
      -Wb,stub_export_macro=ARCOSDAIS<nome do provider>PROVIDER_STUB_Export \
      -Wb,stub_export_include=ARCOSDAIS<nome do provider>Provider_stub_export.h \
      -Wb,skel_export_macro=ARCOSDAIS<nome do provider>PROVIDER_SVNT_Export \
      -Wb,skel_export_include=ARCOSDAIS<nome do provider>Provider_svnt_export.h \
      -I${TAO_ROOT}/orbsvcs/orbsvcs/ -I${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/ \

   includes += ${TAO_ROOT}/orbsvcs/orbsvcs/ ${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/ \
      ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/ \
      ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/ \
      ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/ \
      ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/ \

   libs += OMGDAIS_DnC_stub ARCOSDAISCommon_DnC_stub ARCOSDAISProviderBase_DnC_stub

   ...

}

project(ARCOSDAIS<nome do provider>Provider_DnC_svnt) : ciao_servant_dnc {
   after += OMGDAIS_DnC_svnt ARCOSDAISDAGroup_DnC_svnt ARCOSDAISDANode_DnC_svnt \
      ARCOSDAISProviderBase_DnC_svnt ARCOSDAIS<nome do provider>Provider_DnC_stub

   libs  += OMGDAIS_DnC_stub OMGDAIS_DnC_svnt \
      ARCOSDAISDAGroup_DnC_stub ARCOSDAISDAGroup_DnC_svnt \
      ARCOSDAISDANode_DnC_stub ARCOSDAISDANode_DnC_svnt \
      ARCOSDAISProviderBase_DnC_stub ARCOSDAISProviderBase_DnC_svnt \
      ARCOSDAIS<nome do provider>Provider_DnC_stub

   idlflags += -Sc \
      -Wb,stub_export_macro=ARCOSDAIS<nome do provider>PROVIDER_STUB_Export \
      -Wb,stub_export_include=ARCOSDAIS<nome do provider>Provider_stub_export.h \
      -Wb,skel_export_macro=ARCOSDAIS<nome do provider>PROVIDER_SVNT_Export \
      -Wb,skel_export_include=ARCOSDAIS<nome do provider>Provider_svnt_export.h \
      -I${TAO_ROOT}/orbsvcs/orbsvcs/ -I${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/ \

  cidlflags -= --
  cidlflags += -I${TAO_ROOT}/orbsvcs/orbsvcs/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/ \
      -I${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/ \--

   includes += ${TAO_ROOT}/orbsvcs/orbsvcs/ ${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/ \
      ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/ \
      ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/ \
      ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/ \
```

```
61         ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/
62
63     ...
64
65 }
66
67 project(ARCOSDAIS<nome do provider>Provider_DnC_exec) : ciao_component_dnc {
68     after += ARCOSDAIS<nome do provider>Provider_DnC_svnt
69
70     libs  += OMGDAIS_DnC_stub OMGDAIS_DnC_svnt \
71         ARCOSDAISDAGroup_DnC_stub ARCOSDAISDAGroup_DnC_svnt \
72         ARCOSDAISDANode_DnC_stub ARCOSDAISDANode_DnC_svnt \
73         ARCOSDAISProviderBase_DnC_stub ARCOSDAISProviderBase_DnC_svnt \
74         ARCOSDAIS<nome do provider>Provider_DnC_stub \
75         ARCOSDAIS<nome do provider>Provider_DnC_svnt \
76
77     includes += ${TAO_ROOT}/orbsvcs/orbsvcs/ ${ARCOS_ROOT}/DataAcquisition/DAIS/OMGDAIS/ \
78         ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDAGroup/ \
79         ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISDANode/ \
80         ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISProviderBase/ \
81         ${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISAccessPoints/
82
83     ...
84
85 }
```

Após a execução dos passos acima, o novo *DAIS Provider* está pronto para uso. A montagem (conexão) desse novo provider com as estruturas internas do ARCOS será apresentada na seção 4.2.3.

4.2.2 Implementando um novo controlador

Os passos requeridos para a implementação de um novo controlador são:

1. Criação do diretório e do arquivo de configuração do MPC do novo controlador: um novo controlador pode ser implementado em qualquer diretório do sistema de arquivos, porém recomenda-se que estes sejam reunidos no diretório *${ARCOS_ROOT}/Control/ARCOSControllers/*. O arquivo de configuração do MPC deve ser gerado através do utilitário '*generate_component_mpc.pl <nome do componente>*'. Sugere-se que os nomes dos diretórios sigam o padrão '*ARCOS<nome do controlador>Controller*' como, por exemplo, *ARCOSPIDController*. O comando *generate_component_mpc.pl* irá gerar o arquivo '*ARCOS<nome do controller>Controller.mpc*' e três arquivos requeridos pelo ACE para questões de portabilidade. Edite o arquivo .mpc gerado de modo a refletir as linhas apresentadas no código 4.2.

2. Criação dos arquivos IDL e CIDL do novo controlador: após as alterações no arquivo .mpc, o arquivo IDL e CIDL do novo controlador deverão ser criados pelo desenvolvedor. Um exemplo de arquivo IDL para um novo controlador foi apresentado na sub-seção 3.2.2.1 e o arquivo CIDL pode ser criado conforme descrito na sub-seção 2.2.1.2.

3. Geração da versão inicial do executor do novo controlador: de forma semelhante à implementação

do *DAIS Provider*, o compilador de arquivos CIDL deve ser executado com o parâmetro '*–gen-exec-impl*'. Sugere-se que, nesse momento, o desenvolvedor execute os comandos '*mwc.pl*' e '*make*' para verificar a linha de execução do *cidlc* e alterá-la para a inclusão do parâmetro '*–gen-exec-impl*'. Esse parâmetro deve ser inserido antes do trecho '– <nome do componente>'. Esse procedimento gerará os arquivos *<nome do componente_ exec.h* e *<nome do componente>_ exec.cpp.*

4. Implementação do novo controlador: os arquivos gerados no passo anterior já contêm uma implementação vazia do método *control()*. O desenvolvedor que utiliza o ARCOS deve, portanto, fornecer uma implementação real desse método, executando a estratégia de controle desejada.

5. Compilação do novo controlador: após a completa implementação do controlador, a compilação pode ser realizada através dos comandos '*mwc.pl*' e '*make*'.

Após a execução dos passos acima, o novo controlador está pronto para uso. A montagem (conexão) desse novo controlador com as estruturas internas do ARCOS será apresentada na seção 4.2.3.

4.2.3 Configurando a montagem do sistema

Conforme apresentado na sub-seção 2.2.1.3, a combinação de componentes de software com o objetivo de compor um novo sistema é realizado através do uso de arquivos XML descritores de implantação. No caso de aplicações baseadas no ARCOS, esse arquivo indicará as instâncias de componentes que serão criadas e como estas serão combinadas de modo a formar a nova aplicação de S&C.

A sub-seção 2.2.1.3 apresenta detalhadamente o funcionamento das *tags* utilizadas nesse arquivo descritor. Nesta sub-seção, será apresentado como essas *tags* são utilizadas para compor um novo sistema distribuído de S&C. O código 4.3 apresenta um exemplo de descritor de implantação para aquisição de dados via CLP e uso de um controlador PID. As indicações a serem realizadas no arquivo descritor de implantação são:

1. Indicação dos receptáculos utilizados: no bloco *realizes* do plano de implantação (vide sub-seção 2.2.1.3) deve ser criado um bloco *port* para cada receptáculo utilizado pelo sistema (linhas 13 a 18).

2. Definição dos tipos de componentes a serem utilizados: cada instância de componente, criada no sistema, pertence a um determinado tipo e estes devem ser previamente definidos, no descritor de implantação, através do bloco *implementation*. Para aplicações baseadas no ARCOS os tipos de componentes a serem definidos são: *DAISServer* (linhas 21 a 26), *DAISDANodeIterator* (linhas 28 a 33), *DAISDANodeHome* (linhas 35 a 40), *DAISDAGroupHome* (linhas 42 a 47), *DAISDAGroupMa-*

Código 4.2: Alterações a serem realizadas no arquivo MPC do novo controlador

```
1  project(ARCOS<nome do controlador>Controller_DnC_stub): ciao_client_dnc {
2     after += ARCOSControllerBase_DnC_stub
3
4     idlflags += -Sc \
5        -Wb,stub_export_macro=ARCOS<nome do controlador>CONTROLLER_STUB_Export \
6        -Wb,stub_export_include=ARCOS<nome do controlador>Controller_stub_export.h \
7        -Wb,skel_export_macro=ARCOS<nome do controlador>CONTROLLER_SVNT_Export \
8        -Wb,skel_export_include=ARCOS<nome do controlador>Controller_svnt_export.h \
9        -I${ARCOS_ROOT}/Control/ARCOSControllers/ARCOSControllerBase/
10
11    includes += ${ARCOS_ROOT}/Control/ARCOSControllers/ARCOSControllerBase/
12
13    libs += ARCOSControllerBase_DnC_stub
14
15    ...
16
17 }
18
19 project(ARCOS<nome do controlador>Controller_DnC_svnt) : ciao_servant_dnc {
20    after += ARCOS<nome do controlador>Controller_DnC_stub
21
22    libs  += ARCOSControllerBase_DnC_stub ARCOSControllerBase_DnC_svnt \
23       ARCOS<nome do controlador>Controller_DnC_stub
24
25    idlflags += -Sc \
26       -Wb,stub_export_macro=ARCOS<nome do controlador>CONTROLLER_STUB_Export \
27       -Wb,stub_export_include=ARCOS<nome do controlador>Controller_stub_export.h \
28       -Wb,skel_export_macro=ARCOS<nome do controlador>CONTROLLER_SVNT_Export \
29       -Wb,skel_export_include=ARCOS<nome do controlador>Controller_svnt_export.h \
30       -I${ARCOS_ROOT}/Control/ARCOSControllers/ARCOSControllerBase/
31
32    cidlflags -= --
33    cidlflags += -I${ARCOS_ROOT}/Control/ARCOSControllers/ARCOSControllerBase/ --
34
35    includes += ${ARCOS_ROOT}/Control/ARCOSControllers/ARCOSControllerBase/
36
37    ...
38
39 }
40
41 project(ARCOS<nome do controlador>Controller_DnC_exec) : ciao_component_dnc {
42    after += ARCOS<nome do controlador>Controller_DnC_svnt
43
44    libs  += ARCOSControllerBase_DnC_stub ARCOSControllerBase_DnC_svnt \
45       ARCOS<nome do controlador>Controller_DnC_stub \
46       ARCOS<nome do controlador>Controller_DnC_svnt \
47
48    includes += ${ARCOS_ROOT}/Control/ARCOSControllers/ARCOSControllerBase/
49
50    ...
51
52 }
```

nager (linhas 49 a 54), *DAISDAGroupClock* (linhas 56 a 61), *DAISDAGroupEntryIterator* (linhas 63 a 68), *DAISDASession* (linhas 70 a 75), *ControlManager* (linhas 77 a 82), *ControlManager-DAISCallback* (linhas 84 a 89) e os dois componentes (implementados pelo desenvolvedor) que especializam os *hot-spots* do ARCOS, nesse exemplo, o *DAISEthernetPLCProvider* (linhas 91 a 96) e o *PIDController* (linhas 98 a 103). Apesar de não serem criadas, estaticamente, instâncias de todos os tipos acima descritos, a presença prévia dessas definições facilita a manipulação, via ReDaC, do plano de implantação, requerida para a criação dinâmica dos componentes DAIS.

3. Criação das instâncias estáticas: a figura 3.4 mostra que os únicos componentes de aquisição de

dados criados estaticamente no momento da implantação do sistema são: o *DAISServer*, o *DAISDANodeHome* e o *DAIS Provider* construído pelo desenvolvedor. Os demais componentes são criados, dinamicamente, à medida em que clientes DAIS realizam as operações de consulta da árvore DAIS e criação de grupos de aquisição. Para as atividades de controle, são instanciados os componentes *ControlManager*, *ControlManagerDAISCallback* e o controlador construído pelo desenvolvedor, e nenhuma outra instância necessita ser criada dinamicamente. As instâncias dos componentes *DAISServer* (linhas 105 a 118), *DAISDANodeHome* (linhas 120 a 133) e o *DAIS Provider* (linhas 135 a 157) são criadas informando, através do atributo *RegisterNaming*, o nome a ser utilizado para registro no Servidor de Nomes. O *DAIS Provider*, em particular, deve ter o atributo *plc_ip_address* ajustado para informar o endereço IP do CLP a ser utilizado (linhas 148 a 156). A instância do componente *ControlManager* é criada nas linhas 159 a 197. Para este componente, são informados: o nome a ser utilizado para registro no Servidor de Nomes (linhas 163 a 171), o nome com o qual o servidor DAIS, a ser utilizado pelo controlador, foi registrado no Servidor de Nomes (linhas 172 a 180), a taxa de aquisição de dados (linhas 181 a 189) e o valor do *setpoint* (linhas 190 a 196). A instância do componente *ControlManagerDAISCallback* é criada nas linhas 199 a 212. Finalmente, a instância do controlador implementado pelo desenvolvedor é criada nas linhas 214 a 248.

4. Conexão dos componentes da montagem: após a criação das instâncias necessárias, devem ser realizadas as conexões entre os componentes internos do ARCOS e entre estes e os componentes criados pelo desenvolvedor. A tabela 3.1 apresenta as conexões necessárias entre os componentes internos do ARCOS, no módulo de aquisição de dados. A figura 3.8 apresenta as conexões necessárias ao módulo de controle. Essas conexões são criadas nas linhas 250 a 310, com destaque para as conexões com os componentes implementados pelo desenvolvedor: o *DAIS Provider* (linhas 250 a 260) e o controlador (linhas 286 a 296).

5. Definição dos artefatos (vide sub-seção 2.2.1.3): para cada tipo de componente presente na montagem, devem ser indicados artefatos técnicos da sua implementação, tais como a biblioteca que contém o componente e a função que permite a criação dos *homes* e dos próprios componentes. Conforme apresentado na sub-seção 2.2.1.3, para cada tipo de componente um par de blocos *artifact* deve ser criado, um para o executor e outro para o *servant*. As linhas 312 a 346 apresentam, resumidamente, a construção desses blocos.

Código 4.3: Exemplo de descritor de implantação para aquisição de dados via CLP e controle PID

```
1 <Deployment:deploymentPlan
2     xmlns:Deployment="http://www.omg.org/Deployment"
3     xmlns:xmi="http://www.omg.org/XMI"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.omg.org/Deployment␣Deployment.xsd">
6     <label>ARCOSDAISServer-DeploymentPlan</label>
```

```
7     <UUID>ARCOSDAISServer_Plan_UUID_0001</UUID>
8     <realizes>
9        <label>ARCOSDAISServer-realizes-cid</label>
10       <UUID>2c56c1cd-d7c7-46d5-9afc-ef096db6ff90</UUID>
11       <specificType></specificType>
12       <supportedType>IDL:ufba.br/ARCOS/DataAcquisition/DAISServer:1.0</supportedType>
13       <port><name>dais_provider                  </name> ... </port>
14       <port><name>dais_dataaccess_node_home_ap   </name> ... </port>
15       <port><name>dais_dataaccess_group_home_ap  </name> ... </port>
16       <port><name>dais_server_ap                 </name> ... </port>
17       <port><name>control_manager_callback_ap    </name> ... </port>
18       <port><name>controller                     </name> ... </port>
19    </realizes>
20
21    <implementation id="ARCOS-DAISServer-mdd">
22       <name>ARCOS-DAISServer-mdd</name>
23       <source><!-- @@ Don't know what goes here --></source>
24       <artifact>ARCOS-DAISServer_exec</artifact>
25       <artifact>ARCOS-DAISServer_svnt</artifact>
26    </implementation>
27
28    <implementation id="ARCOS-DAISDANodeIterator-mdd">
29       <name>ARCOS-DAISDANodeIterator-mdd</name>
30       <source><!-- @@ Don't know what goes here --></source>
31       <artifact>ARCOS-DAISDANodeIterator_exec</artifact>
32       <artifact>ARCOS-DAISDANodeIterator_svnt</artifact>
33    </implementation>
34
35    <implementation id="ARCOS-DAISDANodeHome-mdd">
36       <name>ARCOS-DAISDANodeHome-mdd</name>
37       <source><!-- @@ Don't know what goes here --></source>
38       <artifact>ARCOS-DAISDANodeHome_exec</artifact>
39       <artifact>ARCOS-DAISDANodeHome_svnt</artifact>
40    </implementation>
41
42    <implementation id="ARCOS-DAISDAGroupHome-mdd">
43       <name>ARCOS-DAISDAGroupHome-mdd</name>
44       <source><!-- @@ Don't know what goes here --></source>
45       <artifact>ARCOS-DAISDAGroupHome_exec</artifact>
46       <artifact>ARCOS-DAISDAGroupHome_svnt</artifact>
47    </implementation>
48
49    <implementation id="ARCOS-DAISDAGroupManager-mdd">
50       <name>ARCOS-DAISDAGroupManager-mdd</name>
51       <source><!-- @@ Don't know what goes here --></source>
52       <artifact>ARCOS-DAISDAGroupManager_exec</artifact>
53       <artifact>ARCOS-DAISDAGroupManager_svnt</artifact>
54    </implementation>
55
56    <implementation id="ARCOS-DAISDAGroupClock-mdd">
57       <name>ARCOS-DAISDAGroupClock-mdd</name>
58       <source><!-- @@ Don't know what goes here --></source>
59       <artifact>ARCOS-DAISDAGroupClock_exec</artifact>
60       <artifact>ARCOS-DAISDAGroupClock_svnt</artifact>
61    </implementation>
62
63    <implementation id="ARCOS-DAISDAGroupEntryIterator-mdd">
64       <name>ARCOS-DAISDAGroupEntryIterator-mdd</name>
65       <source><!-- @@ Don't know what goes here --></source>
66       <artifact>ARCOS-DAISDAGroupEntryIterator_exec</artifact>
67       <artifact>ARCOS-DAISDAGroupEntryIterator_svnt</artifact>
68    </implementation>
69
70    <implementation id="ARCOS-DAISDASession-mdd">
71       <name>ARCOS-DAISDASession-mdd</name>
72       <source><!-- @@ Don't know what goes here --></source>
73       <artifact>ARCOS-DAISDASession_exec</artifact>
74       <artifact>ARCOS-DAISDASession_svnt</artifact>
75    </implementation>
76
77    <implementation id="ARCOS-ControlManager-mdd">
78       <name>ARCOS-CONTROLManager-mdd</name>
79       <source><!-- @@ Don't know what goes here --></source>
```

```
80          <artifact>ARCOS-ControlManager_exec</artifact>
81          <artifact>ARCOS-ControlManager_svnt</artifact>
82      </implementation>
83
84      <implementation id="ARCOS-ControlManagerDAISCallback-mdd">
85          <name>ARCOS-ControlManagerDAISCallback-mdd</name>
86          <source><!-- @@ Don't know what goes here --></source>
87          <artifact>ARCOS-ControlManagerDAISCallback_exec</artifact>
88          <artifact>ARCOS-ControlManagerDAISCallback_svnt</artifact>
89      </implementation>
90
91      <implementation id="ARCOS-DAISEthernetPLCProvider-mdd">
92          <name>ARCOS-DAISEthernetPLCProvider-mdd</name>
93          <source><!-- @@ Don't know what goes here --></source>
94          <artifact>ARCOS-DAISEthernetPLCProvider_exec</artifact>
95          <artifact>ARCOS-DAISEthernetPLCProvider_svnt</artifact>
96      </implementation>
97
98      <implementation id="ARCOS-PIDController-mdd">
99          <name>ARCOS-PIDController-mdd</name>
100         <source><!-- @@ Don't know what goes here --></source>
101         <artifact>ARCOS-PIDController_exec</artifact>
102         <artifact>ARCOS-PIDController_svnt</artifact>
103     </implementation>
104
105     <instance id="ARCOS-DAISServer-idd">
106         <name>ARCOS-DAISServer-idd</name>
107         <node>MainNode</node>
108         <implementation>ARCOS-DAISServer-mdd</implementation>
109         <configProperty>
110             <name>RegisterNaming</name>
111             <value>
112                 <type><kind>tk_string</kind></type>
113                 <value>
114                     <string>DataAcquisition/PLC_DAIS_Server/ARCOSPLCDAISServer</string>
115                 </value>
116             </value>
117         </configProperty>
118     </instance>
119
120     <instance id="ARCOS-DAISDANodeHome-idd">
121         <name>ARCOS-DAISDANodeHome-idd</name>
122         <node>MainNode</node>
123         <implementation>ARCOS-DAISDANodeHome-mdd</implementation>
124         <configProperty>
125             <name>RegisterNaming</name>
126             <value>
127                 <type><kind>tk_string</kind></type>
128                 <value>
129                     <string>DataAcquisition/PLC_DAIS_Server/ARCOSDAISDANodeHome</string>
130                 </value>
131             </value>
132         </configProperty>
133     </instance>
134
135     <instance id="ARCOS-DAISEthernetPLCProvider-idd">
136         <name>ARCOS-DAISEthernetPLCProvider-idd</name>
137         <node>MainNode</node>
138         <implementation>ARCOS-DAISEthernetPLCProvider-mdd</implementation>
139         <configProperty>
140             <name>RegisterNaming</name>
141             <value>
142                 <type><kind>tk_string</kind></type>
143                 <value>
144                     <string>DataAcquisition/PLC_DAIS_Server/ARCOSDAISEthernetPLCProvider</string>
145                 </value>
146             </value>
147         </configProperty>
148         <configProperty>
149             <name>plc_ip_address</name>
150             <value>
151                 <type><kind>tk_string</kind></type>
152                 <value>
```

```
153                 <string>10.0.0.2</string>
154             </value>
155         </value>
156     </configProperty>
157 </instance>
158
159 <instance id="ARCOS-ControlManager-idd">
160     <name>ARCOS-ControlManager-idd</name>
161     <node>MainNode</node>
162     <implementation>ARCOS-ControlManager-mdd</implementation>
163     <configProperty>
164         <name>RegisterNaming</name>
165         <value>
166             <type><kind>tk_string</kind></type>
167             <value>
168                 <string>Control/ARCOSControlManager</string>
169             </value>
170         </value>
171     </configProperty>
172     <configProperty>
173         <name>dais_server_name</name>
174         <value>
175             <type><kind>tk_string</kind></type>
176             <value>
177                 <string>DataAcquisition/PLC_DAIS_Server/ARCOSPLCDAISServer</string>
178             </value>
179         </value>
180     </configProperty>
181     <configProperty>
182         <name>sampling_rate</name>
183         <value>
184             <type><kind>tk_ushort</kind></type>
185             <value>
186                 <string>200</string>
187             </value>
188         </value>
189     </configProperty>
190     <configProperty>
191         <name>setpoint</name>
192         <value>
193             <type><kind>tk_double</kind></type>
194             <value><double>80</double></value>
195         </value>
196     </configProperty>
197 </instance>
198
199 <instance id="ARCOS-ControlManagerDAISCallback-idd">
200     <name>ARCOS-ControlManagerDAISCallback-idd</name>
201     <node>MainNode</node>
202     <implementation>ARCOS-ControlManagerDAISCallback-mdd</implementation>
203     <configProperty>
204         <name>RegisterNaming</name>
205         <value>
206             <type><kind>tk_string</kind></type>
207             <value>
208                 <string>Control/ARCOSControlManagerDAISCallback</string>
209             </value>
210         </value>
211     </configProperty>
212 </instance>
213
214 <instance id="ARCOS-PIDController-idd">
215     <name>ARCOS-PIDController-idd</name>
216     <node>MainNode</node>
217     <implementation>ARCOS-PIDController-mdd</implementation>
218     <configProperty>
219         <name>RegisterNaming</name>
220         <value>
221             <type><kind>tk_string</kind></type>
222             <value>
223                 <string>Control/ARCOSPIDController</string>
224             </value>
225         </value>
```

```
226         </configProperty>
227         <configProperty>
228             <name>kp</name>
229             <value>
230                 <type><kind>tk_double</kind></type>
231                 <value><double>0.5</double></value>
232             </value>
233         </configProperty>
234         <configProperty>
235             <name>ki</name>
236             <value>
237                 <type><kind>tk_double</kind></type>
238                 <value><double>0.0</double></value>
239             </value>
240         </configProperty>
241         <configProperty>
242             <name>kd</name>
243             <value>
244                 <type><kind>tk_double</kind></type>
245                 <value><double>0.1</double></value>
246             </value>
247         </configProperty>
248     </instance>
249
250     <connection>
251         <name>DAISServer_DAISProvider</name>
252         <internalEndpoint>
253             <portName>dais_provider</portName><kind>Facet</kind>
254             <instance>ARCOS-DAISEthernetPLCProvider-idd</instance>
255         </internalEndpoint>
256         <internalEndpoint>
257             <portName>dais_provider</portName><kind>SimplexReceptacle</kind>
258             <instance>ARCOS-DAISServer-idd</instance>
259         </internalEndpoint>
260     </connection>
261
262     <connection>
263         <name>DAISServer_DAISDANodeHome</name>
264         <internalEndpoint>
265             <portName>dais_dataaccess_node_home_ap</portName><kind>Facet</kind>
266             <instance>ARCOS-DAISDANodeHome-idd</instance>
267         </internalEndpoint>
268         <internalEndpoint>
269             <portName>dais_dataaccess_node_home_ap</portName><kind>SimplexReceptacle</kind>
270             <instance>ARCOS-DAISServer-idd</instance>
271         </internalEndpoint>
272     </connection>
273
274     <connection>
275         <name>ControlManager_ControlManagerDAISCallback</name>
276         <internalEndpoint>
277             <portName>control_manager_dais_callback_ap</portName><kind>Facet</kind>
278             <instance>ARCOS-ControlManagerDAISCallback-idd</instance>
279         </internalEndpoint>
280         <internalEndpoint>
281             <portName>control_manager_dais_callback_ap</portName><kind>SimplexReceptacle</kind>
282             <instance>ARCOS-ControlManager-idd</instance>
283         </internalEndpoint>
284     </connection>
285
286     <connection>
287         <name>ControlManager_PIDController</name>
288         <internalEndpoint>
289             <portName>controller</portName><kind>Facet</kind>
290             <instance>ARCOS-PIDController-idd</instance>
291         </internalEndpoint>
292         <internalEndpoint>
293             <portName>controller</portName><kind>SimplexReceptacle</kind>
294             <instance>ARCOS-ControlManager-idd</instance>
295         </internalEndpoint>
296     </connection>
297
298     <connection>
```

```
299         <name>controldata_event_connection</name>
300         <internalEndpoint>
301             <portName>control_data</portName>
302             <kind>EventPublisher</kind>
303             <instance>ARCOS-ControlManagerDAISCallback-idd</instance>
304         </internalEndpoint>
305         <internalEndpoint>
306             <portName>control_data</portName>
307             <kind>EventConsumer</kind>
308             <instance>ARCOS-ControlManager-idd</instance>
309         </internalEndpoint>
310     </connection>
311
312     <artifact id="ARCOS-DAISServer_exec">                    ... </artifact>
313     <artifact id="ARCOS-DAISServer_svnt">                    ... </artifact>
314
315     <artifact id="ARCOS-DAISDANodeIterator_exec">            ... </artifact>
316     <artifact id="ARCOS-DAISDANodeIterator_svnt">            ... </artifact>
317
318     <artifact id="ARCOS-DAISDANodeHome_exec">                ... </artifact>
319     <artifact id="ARCOS-DAISDANodeHome_svnt">                ... </artifact>
320
321     <artifact id="ARCOS-DAISDAGroupHome_exec">               ... </artifact>
322     <artifact id="ARCOS-DAISDAGroupHome_svnt">               ... </artifact>
323
324     <artifact id="ARCOS-DAISDAGroupManager_exec">            ... </artifact>
325     <artifact id="ARCOS-DAISDAGroupManager_svnt">            ... </artifact>
326
327     <artifact id="ARCOS-DAISDAGroupClock_exec">              ... </artifact>
328     <artifact id="ARCOS-DAISDAGroupClock_svnt">              ... </artifact>
329
330     <artifact id="ARCOS-DAISDAGroupEntryIterator_exec">      ... </artifact>
331     <artifact id="ARCOS-DAISDAGroupEntryIterator_svnt">      ... </artifact>
332
333     <artifact id="ARCOS-DAISDASession_exec">                 ... </artifact>
334     <artifact id="ARCOS-DAISDASession_svnt">                 ... </artifact>
335
336     <artifact id="ARCOS-ControlManager_exec">                ... </artifact>
337     <artifact id="ARCOS-ControlManager_svnt">                ... </artifact>
338
339     <artifact id="ARCOS-ControlManagerDAISCallback_exec">    ... </artifact>
340     <artifact id="ARCOS-ControlManagerDAISCallback_svnt">    ... </artifact>
341
342     <artifact id="ARCOS-DAISEthernetPLCProvider_exec">       ... </artifact>
343     <artifact id="ARCOS-DAISEthernetPLCProvider_svnt">       ... </artifact>
344
345     <artifact id="ARCOS-PIDController_exec">                 ... </artifact>
346     <artifact id="ARCOS-PIDController_svnt">                 ... </artifact>
347
348 </Deployment:deploymentPlan>
```

Após a construção do descritor de implantação, a nova montagem pode ser implantada segundo os passos descritos na seção 2.5.1.1 (código 2.14). Vale ressaltar que a implantação de componentes em diferentes nós de uma rede pode facilmente ser alcançada através dos serviços disponibilizados pelo DAnCE. Cada nó que hospedará componentes deverá executar o objeto *NodeManager* e ter o endereço deste indicado no mapa de nós (apresentado no código 2.15) da máquina que realiza a implantação. O nó no qual cada componente será implantado é indicado na *tag node* do bloco *instance* do componente em questão. Na versão 0.5.1 do CIAO, o objeto *RepositoryManager* do DAnCE está ainda em desenvolvimento, o que requer que as bibliotecas de cada componente sejam manualmente copiadas para o diretório *${ACE_ROOT}/lib* do nó no qual será implantado. Entretanto, a implantação e a conexão de *ports* já é realizada de forma remota e, nas versões futuras do CIAO, o uso do *RepositoryManager* disponibilizará um ambiente flexível e escalável para configuração e implantação de sistemas baseados em componentes.

A construção do descritor de implantação apresentado acima é uma tarefa enfadonha e sujeita a erros. Além disso, grande parte desse descritor refere-se à instanciação e conexão dos componentes internos do ARCOS. De fato, somente duas instâncias e duas conexões precisam ser criadas para adequar o ARCOS a uma situação específica de S&C. Com o objetivo de facilitar os passos requeridos para esta adequação, o ARCOS disponibiliza o *ARCOS Assembly Tool*, descrito a seguir. Exemplos completos de descritores de implantação para aplicações baseadas no ARCOS podem ser encontrados no diretório *${ARCOS_ROOT}/descriptors*.

4.3 ARCOS Assembly Tool

O *ARCOS Assembly Tool* (AST) é uma ferramenta que facilita a construção de novas aplicações baseadas no ARCOS, através da execução automática de grande parte dos passos apresentados na seção 4.2. O AST disponibiliza operações para: criação de um novo *DAIS Provider*, criação de um novo controlador, edição e compilação destes novos componentes e implantação da montagem.

O AST, apresentado nas figuras 4.2, 4.3, 4.4 e 4.5, realiza de forma automática a maioria dos passos necessários para a especialização do ARCOS. A partir dos botões destacados na figura 4.2, o desenvolvedor pode facilmente criar novos *DAIS Providers* e controladores, editá-los, compilá-los e realizar a implantação da nova montagem. Todas essas operações podem ser realizadas diretamente no AST.

Para a criação de um novo *DAIS Provider*, o desenvolvedor pressiona o botão "*Create DAIS Provider*", informa o nome do novo componente e o AST irá gerar os arquivos IDL, CIDL, de configuração do MPC e a versão inicial da implementação do executor. Um novo controlador pode ser criado, de forma semelhante, pressionando o botão "*Create Controller*". Todos os passos descritos nas sub-seções 4.2.1 e 4.2.2, com exceção da implementação do executor, são realizados automaticamente pelo AST. Com os botões "*Compile*" e "*Deploy*", o desenvolvedor pode, respectivamente, compilar os novos componentes e implantar a nova montagem.

O AST apresenta quatro visões (*views*) da especialização: *Assembly View* (figura 4.2), *DAIS Provider View* (figura 4.3), *Controller View* (figura 4.4) e *Deployment Descriptor View* (figura 4.5). O *Assembly View* apresenta uma representação gráfica da nova especialização, apresentando os pontos de conexão dos componentes criados pelo desenvolvedor com os componentes internos do ARCOS. O *DAIS Provider View* permite a visualização e edição do código-fonte do novo *provider*. Alterações podem ser diretamente realizadas nessa visão, para posterior compilação. O *Controller View* permite a visualização e edição do código-fonte do novo controlador, também permitindo alterações para posterior compilação. O *Deployment Descriptor View* apresenta o descritor XML gerado pelo AST. Esse descritor já contém todas as configurações necessárias para a implantação (indicadas na sub-seção 4.2.3), porém alterações podem ser realizadas de modo a atender outras exigências da aplicação em questão.

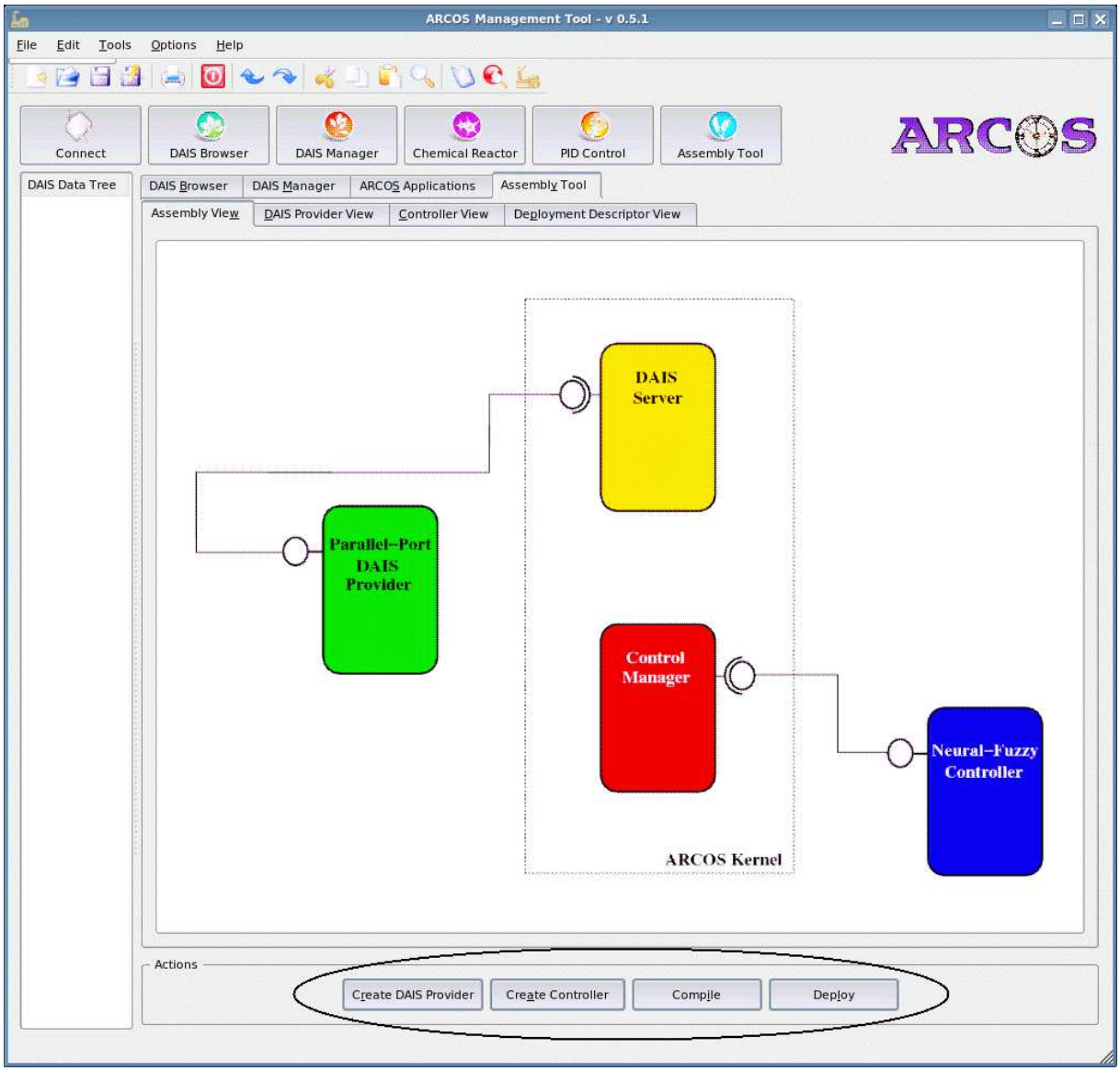


Figura 4.2: *Assembly View* do *ARCOS Assembly Tool*

Os arquivos IDL, CIDL, de configuração do MPC, versão inicial do executor e descritor de implantação são automaticamente gerados pelo AST. Essa geração é realizada a partir de arquivos *templates* que definem variáveis a serem substituídas por valores informados pelo usuário da ferramenta. O código 4.4 apresenta o *template* utilizado para a geração do arquivo IDL de um novo *DAIS Provider*. As variáveis *CapProviderName* e *ProviderName* serão substituídas pelo nome do *provider* informado pelo usuário do AST, sendo o *CapProviderName* substituído pelo nome do *provider*, transformado em maiúsculas.

Uma abordagem semelhante é utilizada para a geração dos demais arquivos. Para o processamento da substituição das variáveis, foi implementada uma classe denominada *ARCOS Template Engine*, apresentada na sub-seção 4.3.1.

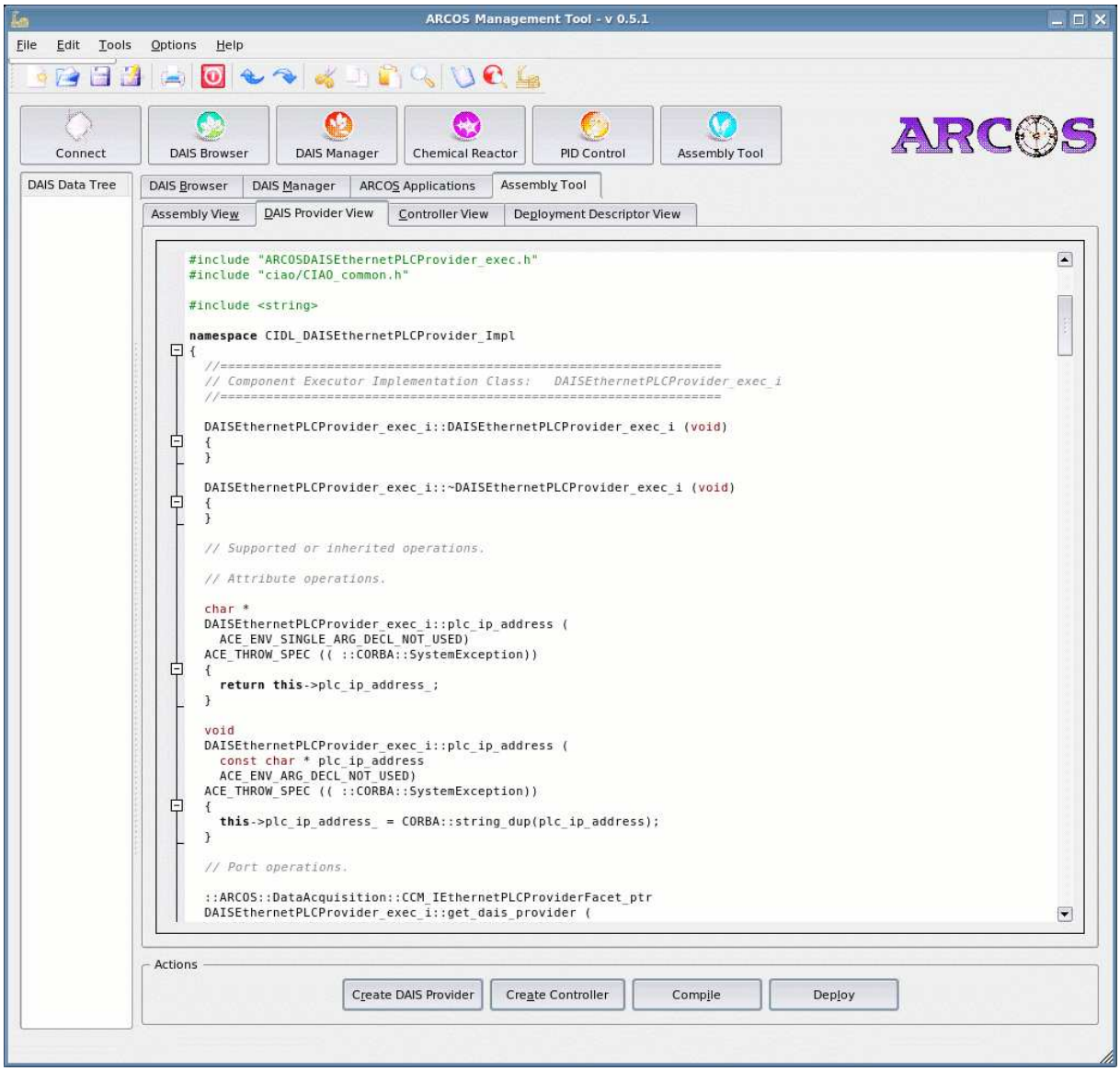


Figura 4.3: *Provider View* do *ARCOS Assembly Tool*

4.3.1 ARCOS Template Engine

A classe *ARCOS Template Engine* é responsável pela geração dos arquivos necessários para a especialização do ARCOS, a partir de *templates* que definem variáveis a serem substituídas. Conforme apresentado na seção 4.3, um *template* é um arquivo texto convencional que define variáveis na forma '*${nome da variável}*'. O objetivo do *ARCOS Template Engine* é fornecer uma classe onde: *i*) variáveis podem ser definidas e ter seus valores informados e *ii*) arquivos *template* podem ser processados, originando arquivos reais.

O código 4.5 apresenta a classe *ARCOS Template Engine*. Os métodos *setVariable()* (linha 17) e *removeVariable()* (linha 18) são utilizados, respectivamente, para incluir e remover uma variável do *template engine*. O método *setVariable()* ao mesmo tempo em que cria a variável cujo nome encontra-se no parâmetro *var*, realiza a sua inicialização com o valor definido no parâmetro *value*. O método

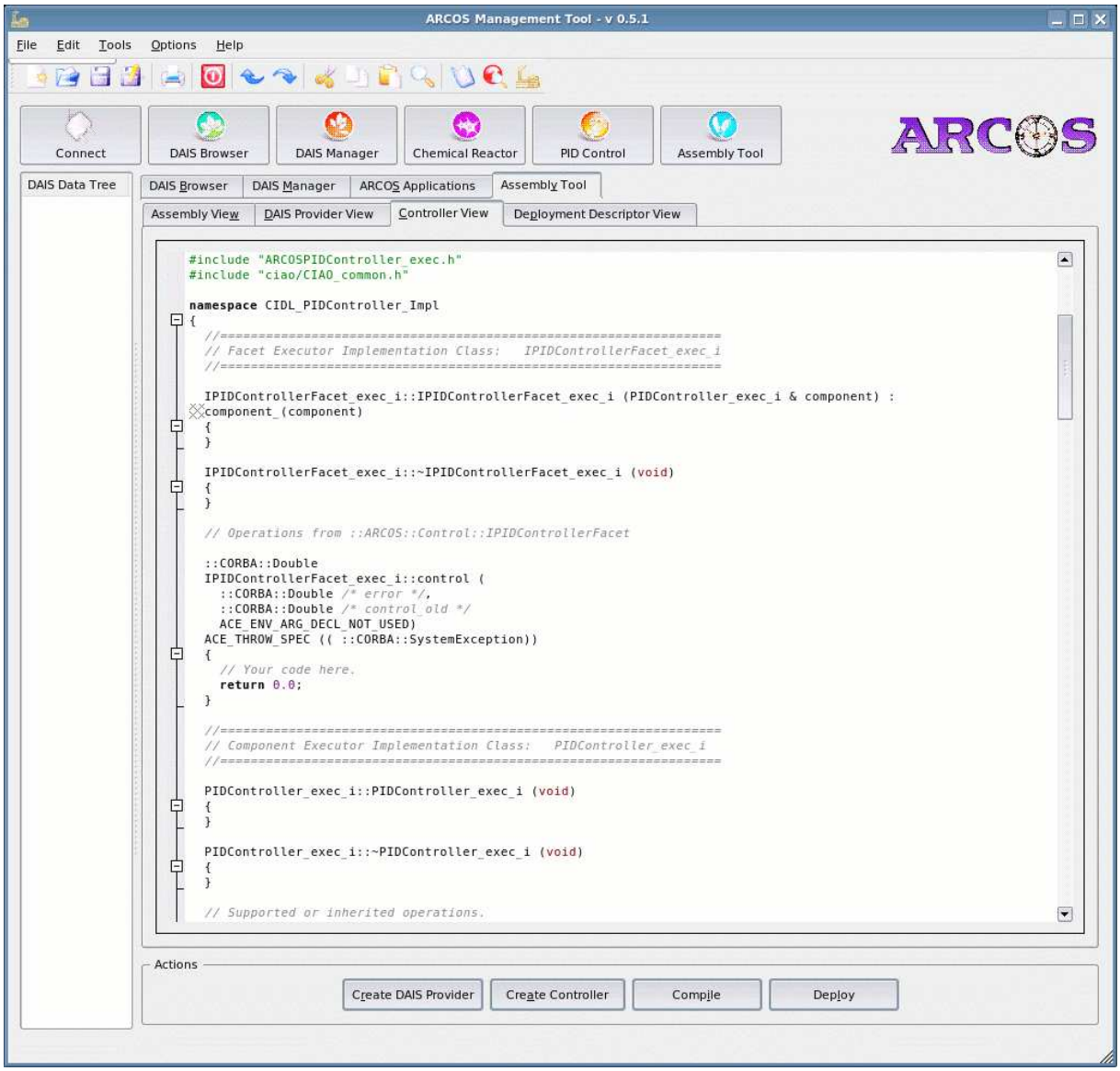


Figura 4.4: *Controller View* do *ARCOS Assembly Tool*

getVariable() (linha 19) retorna o valor definido para a variável cujo nome encontra-se no parâmetro *var*. O método *clean()* (linha 20) apaga todas as variáveis definidas no *template engine*. O método *process()* (linha 21) realiza a transformação de um arquivo *template* em um arquivo real. As variáveis presentes no arquivo especificado pelo parâmetro *input_file* serão substituídas pelos valores previamente definidos, sendo o resultado deste processamento gravado no arquivo especificado pelo parâmetro *output_file*.

O *ARCOS Assembly Tool* faz uso extensivo do *ARCOS Template Engine* para a geração de arquivos IDL, CIDL, de configuração do MPC, versão inicial do executor e descritor XML de implantação. Os *templates* para todos esses arquivos encontram-se no diretório *${ARCOS_ROOT}/Supervision/templates/*.

Código 4.4: *Template* utilizado pelo AST para criação de um novo *DAIS Provider*

```
#ifndef ARCOSDAIS${CapProviderName}PROVIDER_IDL
#define ARCOSDAIS${CapProviderName}PROVIDER_IDL

#pragma prefix "ufba.br"

#include <Components.idl>
#include <ARCOSDAISProviderBase.idl>

module ARCOS
{
    module DataAcquisition
    {
        interface I${ProviderName}ProviderFacet : IDAISProviderBaseFacet
        {
        };
        component DAIS${ProviderName}Provider
        {
            provides ::ARCOS::DataAcquisition::I${ProviderName}ProviderFacet dais_provider;
        };
        home DAIS${ProviderName}ProviderHome manages DAIS${ProviderName}Provider
        {
        };
    };
};
#endif /* ARCOSDAIS${CapProviderName}PROVIDER_IDL */
```

Código 4.5: Classe que implementa o *ARCOS Template Engine*

```
#ifndef ARCOSTEMPLATEENGINE_H
#define ARCOSTEMPLATEENGINE_H

#include "ARCOSUTILS_export.h"

#include <string>
#include <map>

using namespace std;

namespace ARCOS
{
class ARCOSUTILS_Export TemplateEngine
{
    public:

        void   setVariable    (string var, string value);
        void   removeVariable (string var);
        string getVariable    (string var);
        void   clean          ();
        void   process        (string input_file, string output_file);

    private:

        map <string, string> var_map_;
   };
}

#endif /* ARCOSTEMPLATEENGINE_H */
```

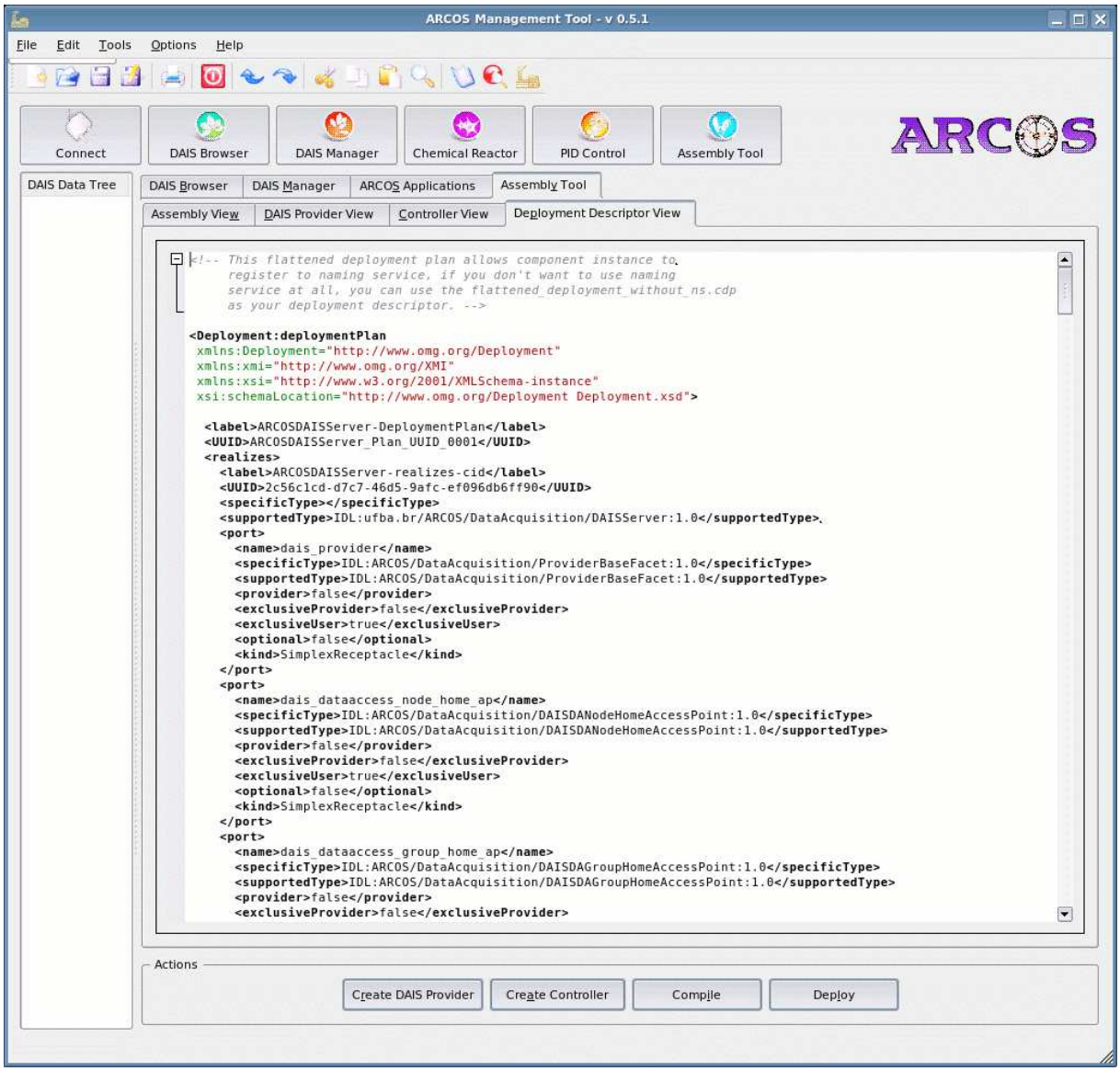


Figura 4.5: *Descriptor View* do *ARCOS Assembly Tool*

4.4 Aplicações exemplo

Com o objetivo de validar a implementação do ARCOS e verificar a sua efetiva utilização em ambientes de S&C, duas aplicações exemplo foram construídas neste projeto: supervisão de um reator químico e controle PID para piloto automático. Essas aplicações apresentam como o ARCOS pode ser utilizado para implementação das atividade de aquisição de dados, controle e supervisão.

4.4.1 Experimento 1: supervisão de um reator químico

O experimento de supervisão de um reator químico, desenvolvido neste projeto, visa a verificação da correta implementação dos serviços de aquisição de dados, disponibilizadas pelo ARCOS. Este experimento

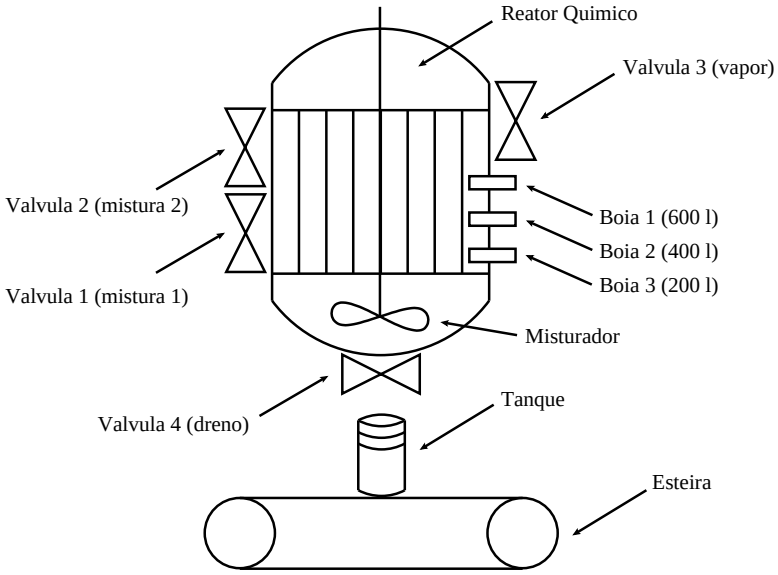


Figura 4.6: Instrumentação do reator químico

é caracterizado pela supervisão do reator químico, monitorando os estados dos sensores e atuadores, e não há presença de malhas de controle fechadas. Os serviços de controle, disponibilizados pelo ARCOS, são avaliados no experimento 2.

4.4.1.1 Descrição do processo

Neste experimento, um reator químico simulado produz um determinado produto através da execução de um processo em batelada[2] [35]. A figura 4.6 apresenta um diagrama com a instrumentação utilizada no experimento. O processo de manufatura é realizado através da execução dos passos descritos a seguir.

1. O processo é iniciado com o posicionamento do tanque sobre a válvula dreno. Satisfeita esta condição, inicia-se o aquecimento do reator químico através da injeção de vapor, causada pela abertura da válvula 3.

2. Atingida a temperatura de 100°C, a válvula 3 é fechada e a abertura da válvula 1 permite a entrada da mistura 1 no reator químico. A válvula 1 permanece aberta até que 400 litros da mistura 1 estejam presentes no tanque, situação esta indicada pelo acionamento das bóias 3 e 2.

3. Após o acionamento da bóia 2, a válvula 1 é fechada e a abertura da válvula 2 permite a entrada da mistura 2 no reator químico. A válvula 2 permanece aberta até que 200 litros da mistura 2 estejam presentes no tanque, situação esta indicada pelo acionamento da bóia 1. Durante a entrada da

[2] Do inglês '*batch process*'. Processo caracterizado pela produção em lotes, ou seja, uma determinada quantidade do produto é produzida através da execução de um procedimento com início e fim bem definidos. Esse procedimento é repetido para a produção de um novo lote.

mistura 2, a temperatura do reator deve ser mantida em 100°C ±10%. Sempre que a temperatura cair abaixo dessa faixa, o processo é interrompido e a temperatura do reator é reajustada, através da abertura da válvula 3 (válvula de entrada de vapor).

4. Após o acionamento da bóia 1, o reator químico encontra-se com a presença das duas misturas e a temperatura deve ser elevada para 300°C, operação esta realizada com a abertura da válvula 3. Atingida essa temperatura, o misturador do reator químico é acionado por 20 segundos e a temperatura deve ser mantida em 300°C ±10%. Sempre que a temperatura cair abaixo dessa faixa, o processo é interrompido e a temperatura do reator é reajustada, através da re-abertura da válvula 3.

5. Com o desligamento do misturador, a abertura da válvula 4 (dreno) permite o escoamento do produto em direção ao tanque. A válvula 4 permanece aberta até que as bóias 1, 2 e 3 sejam, nesta ordem, desacionadas. O processo termina com a saída do tanque da posição sobre a válvula dreno. Uma nova batelada se inicia com o posicionamento de um novo tanque.

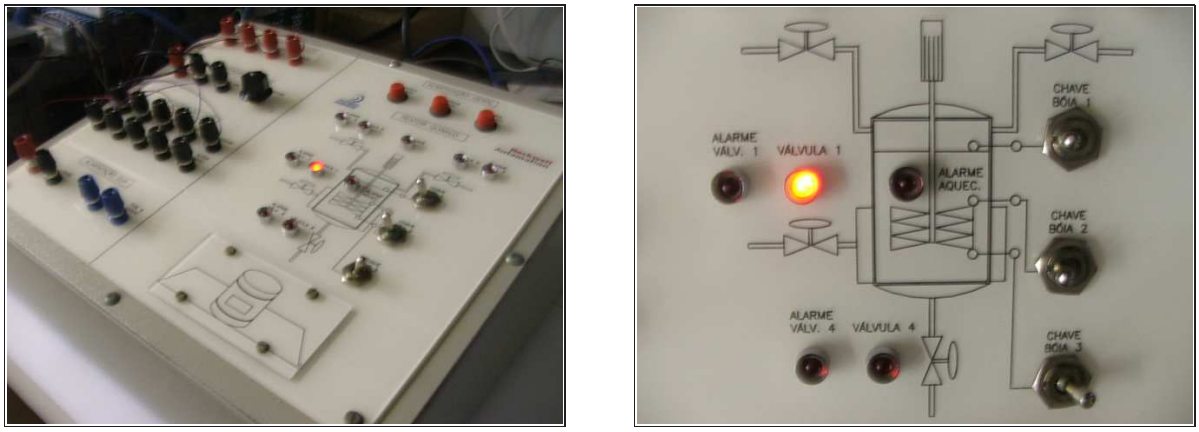
4.4.1.2 Implementação

O experimento foi implementado com a utilização de um *kit* didático para simulação do reator químico, um CLP *Allen-Bradley SLC 5/05* para automação do processo e um *notebook* para execução do ARCOS. A rede de comunicação foi implementada, de forma simples, através de um cabo *crossover* entre o *notebook* e o CLP utilizado.

O reator químico a ser supervisionado foi simulado por um *kit* didático, desenvolvido pela *Rockwell Automation* e ilustrado na figura 4.7.a. As bóias foram simuladas por chaves a serem comandadas manualmente e a temperatura é representada por um potenciômetro que ajusta um valor proporcional de tensão. A abertura e fechamento das válvulas, bem como o acionamento do misturador são indicados através de *leds* (apresentados na figura 4.7.a) no painel do *kit* didático. As operações realizadas no kit para simular o processo descrito na sub-seção 4.4.1.1 são ilustradas na figura 4.8 (imagens *a* a *p*).

A automação do processo em batelada foi implementado em um CLP *Allen-Bradley SLC 5/05*, apresentado na figura 4.7.b. Nesse CLP, foram utilizados: um cartão com 16 entradas digitais (para as bóias e sensor de posicionamento do tanque), um cartão com 16 saídas digitais (para acionamento das válvulas e do misturador) e um cartão de 2 entradas analógicas (sendo somente uma utilizada para o potenciômetro que representa a temperatura) e 2 saídas analógicas (não utilizadas).

As ligações elétricas realizadas entre o CLP e o Kit Didático estão ilustradas na figura 4.9. As chaves que representam as três bóias e o sensor que detecta a presença do tanque são ligadas às entradas do cartão digital, enquanto os *leds* indicadores das válvulas e do acionamento do misturador são ligados às



(a) *Kit* didático

(b) CLP

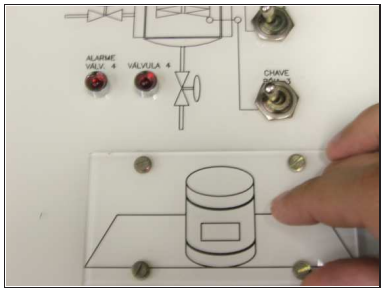
Figura 4.7: *Kit* didático e CLP utilizados no experimento de supervisão do reator químico

saída do cartão digital. O potenciômetro que simula a variação da temperatura é ligado à entrada do cartão analógico. A saída 24V da fonte do CLP deve ser ligada à entrada VDC1 do cartão de saída digital, enquanto o fio *comum* deve ser ligado às entradas correspondentes dos três cartões utilizados. A segunda entrada do cartão analógico deve ser curto-circuitada para indicar a sua não utilização.

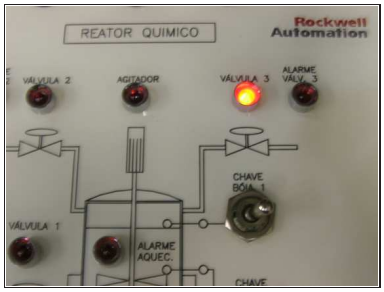
O CLP *Allen-Bradley SLC 5/05* executa o programa que implementa o processo descrito na sub-seção 4.4.1.1. Esse programa, desenvolvido na linguagem LADDER, é apresentado no apêndice F.

O experimento acima descrito é caracterizado pela execução de atividades de automação e pela necessidade de aquisição e monitoramento dos dados da planta. Nesse experimento não há implementação de malhas fechadas de controle[3]. Para que o ARCOS seja utilizado para a implementação de um sistema supervisório para esse reator químico é necessário: *i*) identificar as distintas tecnologias de aquisição de dados utilizadas, *ii*) construir um *DAIS Provider* para cada tecnologia e *iii*) configurar e implantar o servidor DAIS que disponibilizará os dados da planta. A tecnologia de aquisição de dados adotada,

[3] Apesar da possibilidade de uso de um controlador PID para manter a temperatura do reator em um valor constante.

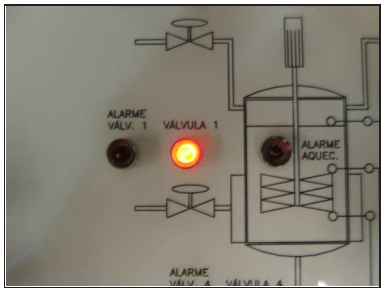


(a) Posicionamento do tanque sobre o reator químico

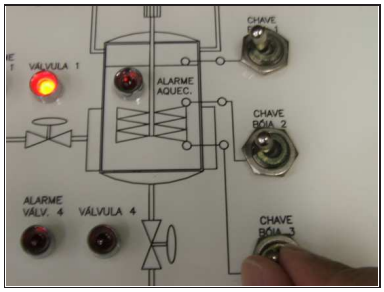


(b) Abertura da válvula de vapor

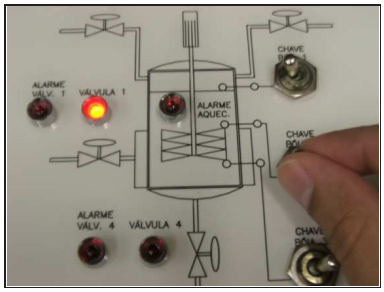
(c) Simulação do aumento da temperatura para 100°C



(d) Abertura da válvula 1

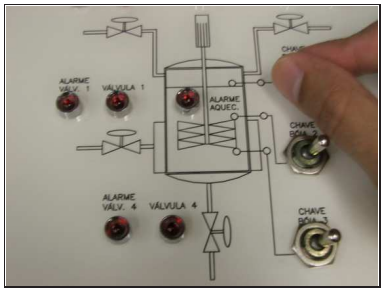


(e) Simulação do acionamento da bóia 3

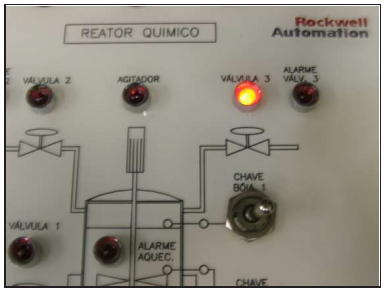


(f) Simulação do acionamento da bóia 2

(g) Abertura da válvula 2



(h) Simulação do acionamento da bóia 1



(i) Nova abertura da válvula de vapor

Figura 4.8: Execução, no *kit* didático, do procedimento do reator químico

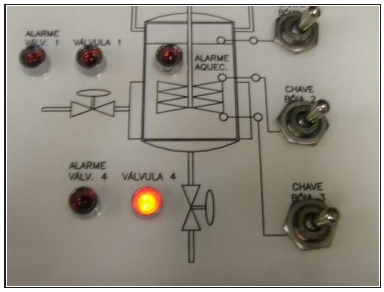
neste experimento, foi a utilização de um CLP conectado a uma rede *Ethernet*, o que requer a implementação, pelo desenvolvedor, de um *DAIS Provider* com esta capacidade de comunicação (figura 4.10). Esse *provider*, denominado *DAISEthernetCLPProvider*, será apresentado a seguir. Caso outras tecnologias de aquisição estivessem sendo utilizadas, tais como porta paralela ou redes industriais, *providers* correspondentes deveriam ser implementados.

DAIS Ethernet CLP Provider

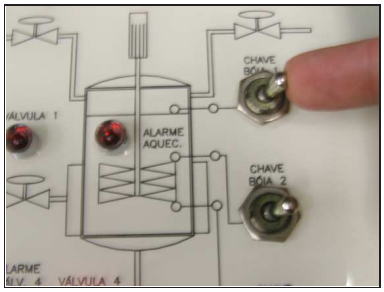
Conforme apresentado na sub-seção 4.2.1, a implementação de um novo *DAIS Provider* requer a execução de duas atividades: projeto e construção da árvore DAIS e a implementação dos métodos para aquisição e escrita de dados. O projeto da árvore DAIS define uma classificação semântica dos dados a

(j) Simulação do aumento da temperatura para 300°C

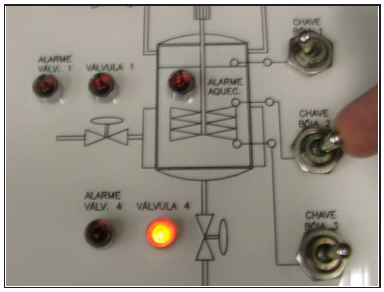
(k) Acionamento do misturador por um tempo de 20s



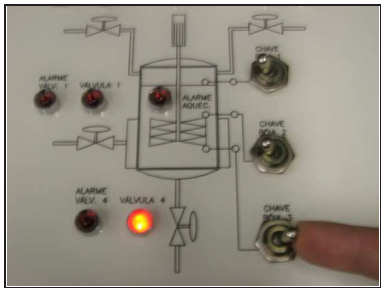
(l) Abertura da válvula dreno



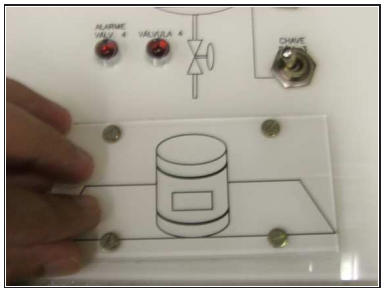
(m) Desacionamento da bóia 1



(n) Desacionamento da bóia 2



(o) Desacionamento da bóia 3



(p) Saída do tanque

Figura 4.8: Execução, no *kit* didático, do procedimento do reator químico (continuação)

serem disponibilizados pelo servidor DAIS conectado ao *provider* em questão, enquanto os métodos de aquisição e escrita realizam a comunicação com os sensores e atuadores, geralmente de forma dependente da tecnologia utilizada nos dispositivos. Para esse experimento, foi implementado o *provider DAISEthernetPLCProvider*, o qual realiza a comunicação com um CLP *Allen-Bradley SLC 5/05* conectado em uma rede *Ethernet*.

A programação de um CLP é geralmente caracterizada pela manipulação dos sinais dos cartões de entrada e saída, bem como dos registradores internos utilizados para simular variáveis auxiliares do tipo *bit*, inteiro, ponto flutuante, contadores, *timers* e de outros tipos que indicam o estado do CLP. No projeto da árvore disponibilizada pelo *DAISEthernetPLCProvider*, criou-se um ramo para cada tipo de dado possível de ser manipulado no CLP. A tabela 4.1 apresenta os ramos presentes na árvore e os tipos

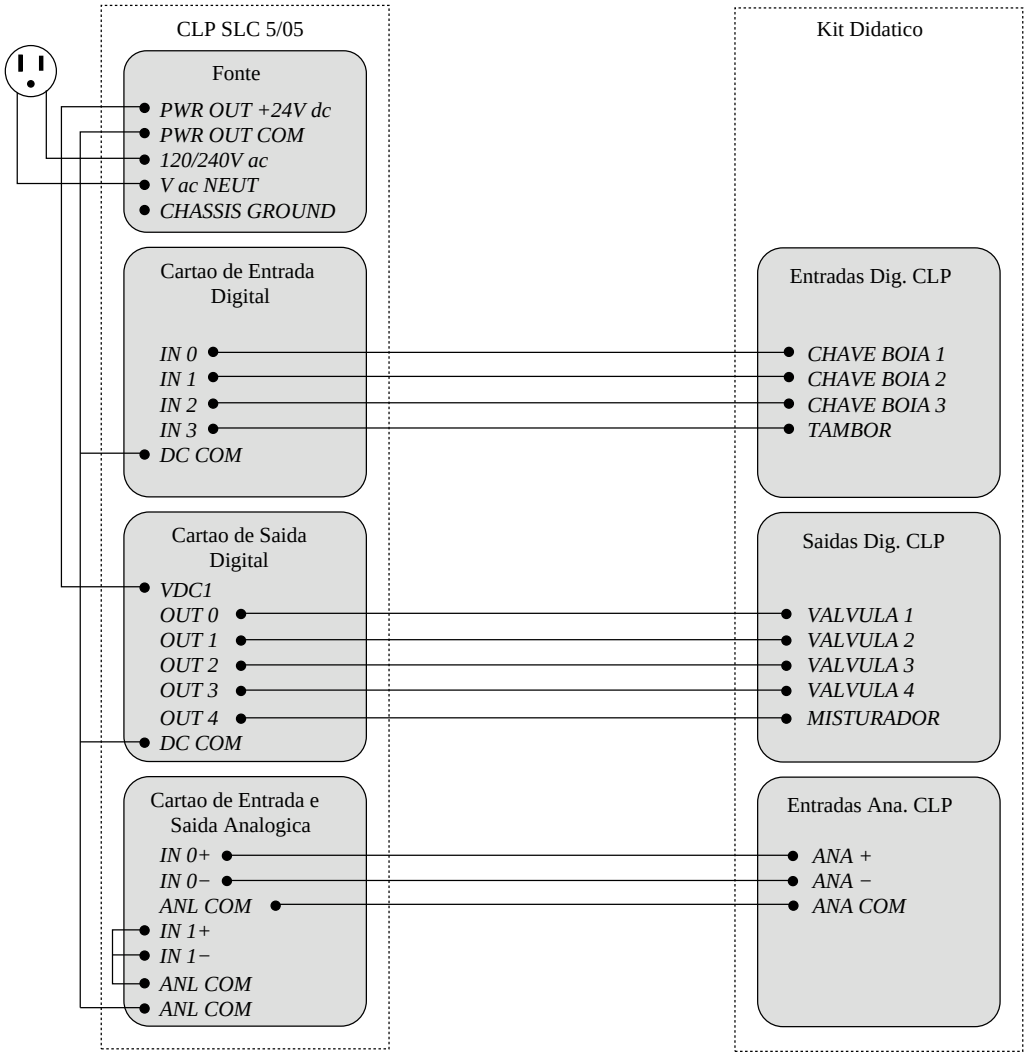


Figura 4.9: Ligações elétricas entre o *kit* didático e o CLP

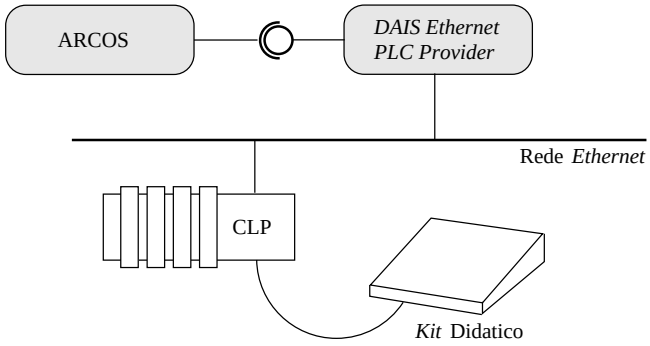


Figura 4.10: Participantes do experimento do reator químico

RAMO	TIPO DE DADO AGRUPADO
DI*n*	Sinais do cartão de entrada digital (*Digital Input*) instalado no *slot n* do CLP
DO*n*	Sinais do cartão de saída digital (*Digital Output*) instalado no *slot n* do CLP
AIO*n*	Sinais do cartão de entrada e saída analógico (*Analog Input* Output) instalado no *slot n* do CLP
S2	Registradores internos de estado do CLP
B3	Registradores internos para variáveis do tipo *bit*
T4	Registradores internos para variáveis do tipo *timer*
C5	Registradores internos para variáveis do tipo contador
R6	Registradores internos para variáveis do tipo controle
N7	Registradores internos para variáveis do tipo inteiro
F8	Registradores internos para variáveis do tipo ponto flutuante

Tabela 4.1: Ramos da árvore DAIS definida pelo *DAISEthernetPLCProvider*

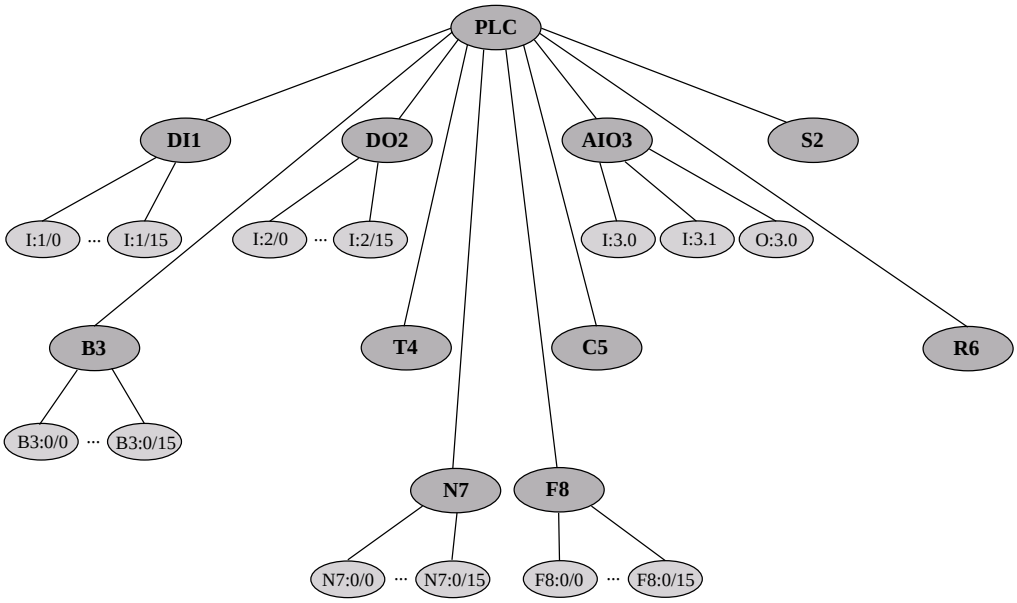


Figura 4.11: Árvore DAIS projetada para o *DAISEthernetPLCProvider*

de dados agrupados por cada um deles.

A figura 4.11 apresenta a árvore DAIS definida pelo *DAISEthernetPLCProvider*. O nó raiz, definido como "PLC", possui os 10 ramos apresentados na tabela 4.1. Essa árvore estará disponível para visualização através das versões web e *desktop* do *DAIS Server Browser*, onde o usuário da ferramenta poderá escolher quais folhas DAIS deverão ser adquiridas. A obtenção do conteúdo dos registradores S2, T4, C5 e R6 não estão atualmente implementados, motivo pelo qual os ramos correspondentes não disponibilizam folhas DAIS. A construção dessa árvore DAIS é realizada pelo método *build_dais_tree()* do componente *DAISEthernetPLCProvider*, disponível no diretório *${ARCOS_ROOT}/DataAcquisition/DAIS/ARCOSDAISProviders/ARCOSDAISEthernetDAISProvider/*.

A comunicação com o *CLP Allen-Bradley SLC 5/05* pode ser realizado de três formas: via porta serial, via rede *Ethernet* ou via rede DH+ (padrão proprietário da *Rockwell Automation*). O *DAISEthernetPLCProvider* faz uso da rede *Ethernet* para comunicação com o CLP, requerendo a conexão do CLP à rede e a configuração do endereço IP a ser utilizado. A comunicação via *Ethernet* foi implementada, no

DAISEthernetPLCProvider, através do uso da biblioteca ABEL (*Allen-Bradley Ethernet Library*) [32]. A ABEL disponibiliza funções, escritas na linguagem C, para a obtenção de dados do CLP a partir da informação do endereço IP e do registrador ou sinal de cartão a ser obtido. O *DAISEthernetCLPProvider* utiliza essas funções para a implementação dos métodos *get_value()* e *set_values()*, requeridos pelo ARCOS. Conforme apresentado na linha 19 do código 3.2, o atributo *plc_ip_address* do componente *DAISEthernetPLCProvider* armazena o endereço IP do CLP ao qual o *provider* deve se conectar. Este endereço IP deve ser informado no arquivo descritor de implantação utilizado (vide código 4.3), conforme apresentado nas linhas 148 a 156.

Supervisão do reator químico

Após a implantação da montagem que utiliza o *DAISEthernetPLCProvider*, a execução dos passos, no *Kit* didático, pode ser monitorada através do *DAIS Server Browser*, disponibilizado pelo ARCOS. A figura 4.12 apresenta a supervisão do reator químico através do *DAIS Server Browser*.

Foram criados quatro grupos de aquisição (somente para fins de agrupamento) com taxa de 200ms: grupo das válvulas, grupo das bóias, um grupo para o misturador e o sensor de posicionamento do tanque e outro grupo para o sensor de temperatura. No grupo das válvulas foram incluídas (através do botão "*Add Leaf as Entry*") as folhas DAIS que representam a válvula de vapor (*OUT 2*), da mistura 1 (*OUT 0*), da mistura 2 (*OUT 1*) e de drenagem (*OUT 3*) (vide ligações elétricas apresentadas na figura 4.9). As folhas correspondentes às outras variáveis foram inseridas nos devidos grupos.

Para cada folha DAIS inserida, um *alias* (apelido) pode ser a ela associado, com o objetivo de facilitar a supervisão daquela folha através de um nome que represente diretamente um objeto do chão-de-fábrica. No caso de aquisição de variáveis analógicas, como por exemplo a temperatura, o botão "*Unit Convertion*" pode ser utilizado para transformar os valores informados pelo CLP (geralmente no intervalo [0, 32767]) em valores que representam diretamente a grandeza sendo monitorada (por exemplo, a temperatura no intervalo [0°C, 500°C]).

O *DAIS Server Browser* representa uma solução genérica e interoperável para a supervisão de dados disponibilizados por um servidor DAIS, possibilitando a visualização da árvore DAIS definida, criação de grupos de aquisição e inserção de folhas DAIS nestes grupos. Apesar desses benefícios e da utilização de *aliases* para auxiliar a supervisão, o *DAIS Server Browser* não apresenta facilidades voltadas para um caso particular de S&C. Pelo fato de ser um cliente genérico para servidores DAIS, nada pode ser assumido em relação à semântica dos dados sendo disponibilizados pelo servidor em questão.

Portanto, com o objetivo de proporcionar uma supervisão mais especializada do problema em questão, foi desenvolvido, nesta dissertação, um supervisório específico para o experimento do reator químico. Esse supervisório, apresentado na figura 4.13, monitora os elementos presentes na planta através de animações 3D, executadas na cena apresentada.

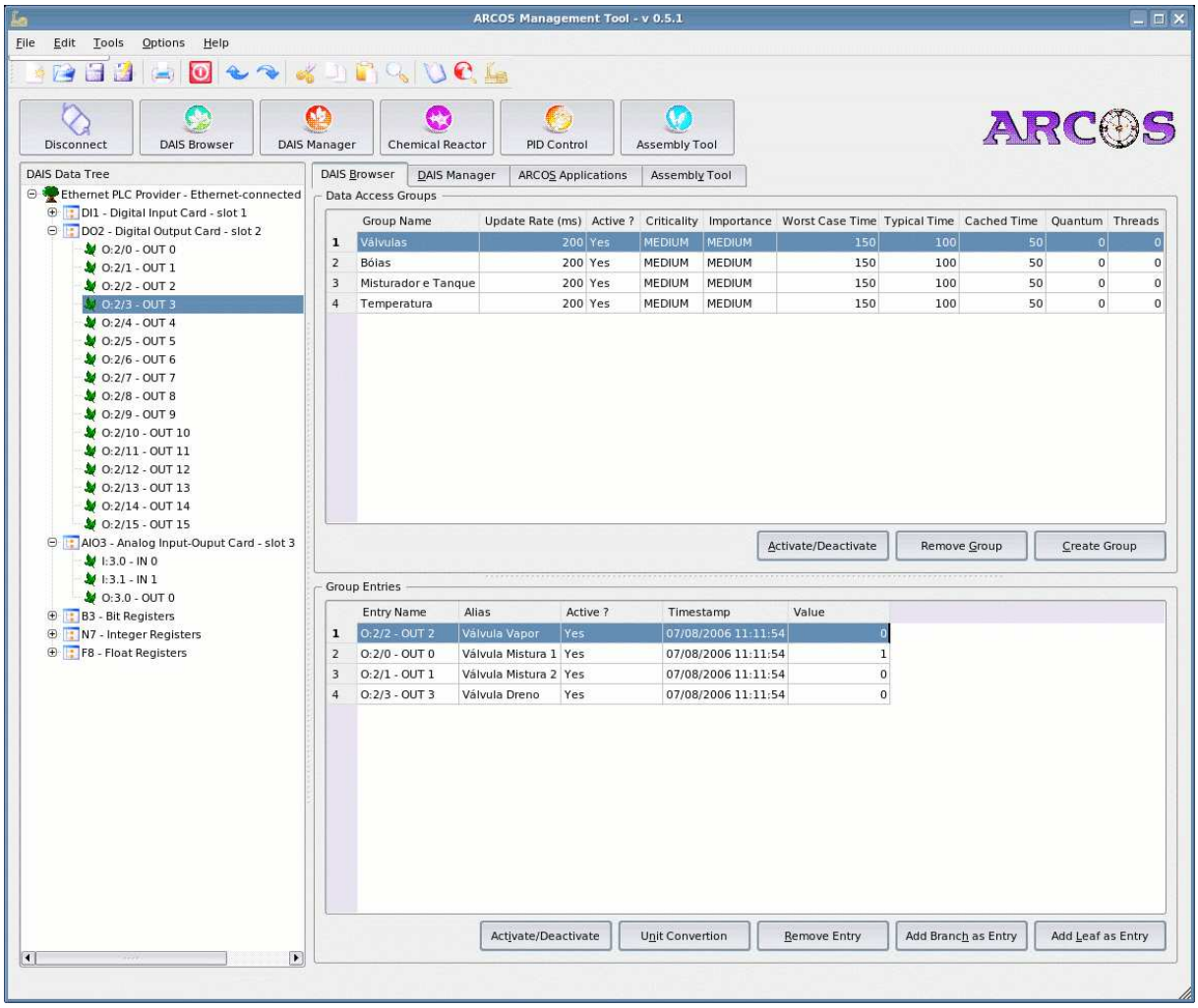


Figura 4.12: Supervisão do reator químico através do *DAIS Server Browser*

Antes do início da supervisão, o usuário deve indicar as folhas DAIS que estão associadas às válvulas, bóias, misturador, sensor do tanque e sensor de temperatura. Os quatro grupos que foram utilizados no *DAIS Server Browser* são automaticamente criados nesse supervisório específico. Para cada sensor/atuador, o usuário deve selecionar a folha DAIS que o representa e pressionar o botão correspondente (no grupo "*DAIS Leaves Selection*"). Com essa ação, a folha é incluída no grupo de aquisição e associada às animações a serem executadas. Por exemplo, uma mudança, de 0 para 1, do valor da folha que representa a válvula de vapor, dispara a animação da abertura da válvula correspondente. Com essa abordagem, as operações realizadas no *kit* didático são imediatamente refletidas no supervisório específico. A figura 4.13 apresenta o momento em que a temperatura encontra-se a 100°C, a válvula da mistura 1 está aberta e a bóia 3 é acionada.

A modelagem 3D de toda a cena da fábrica, bem como a construção das animações foram realizadas no *Blender* [24], um software livre para modelagem, visualização, animação e desenvolvimento de jogos. Esse supervisório específico para o reator químico está disponível no *ARCOS Management Tool*, no *tab ARCOS Applications*.

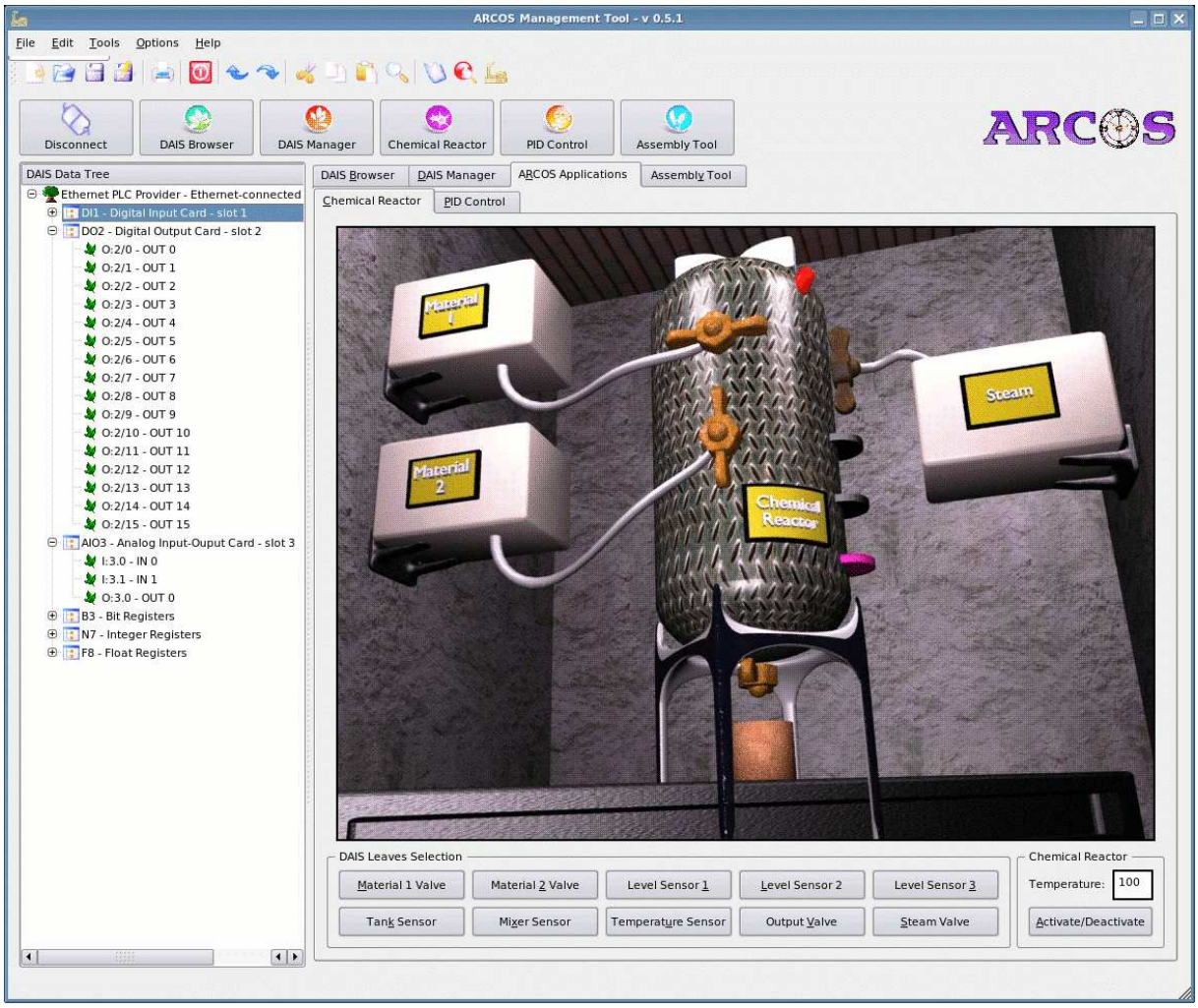


Figura 4.13: Supervisão do reator químico através do supervisório específico

Este experimento verifica a facilidade com que sistemas de S&C podem ser construídos, quando utilizando o ARCOS. A despeito dos detalhes intrínsecos à construção do supervisório específico, a monitoração do reator é construída através da execução dos cinco passos descritos na seção 4.2. O uso do *ARCOS Assembly Tool*, como uma ferramenta de apoio ao desenvolvimento, torna o processo mais produtivo e seguro.

4.4.2 Experimento 2: controle PID para piloto automático

O experimento 1 validou a utilização facilitada do módulo de aquisição de dados disponibilizado pelo ARCOS, sem entretanto implementar malhas fechadas de controle. Neste segundo experimento, o ARCOS será utilizado na construção de uma malha fechada para controle de velocidade de um veículo, além de também executar os procedimentos de aquisição. Serão apresentados os objetivos da operação de controle, o modelo matemático utilizado para a simulação do veículo e a implementação realizada com base no ARCOS.

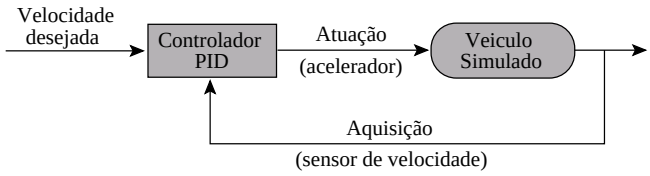


Figura 4.14: Controle PID da velocidade do veículo

4.4.2.1 Descrição da malha de controle

Neste experimento, deseja-se manter a velocidade de um veículo em torno de um determinado valor (*set-point*), mesmo na presença de fatores que provocariam distúrbios nesta velocidade. Conforme apresentado na figura 4.14, foi utilizado um controlador PID convencional que recebe os dados da planta (veículo), faz o cálculo da ação de controle e solicita as operações de atuação. A aquisição é realizada através de um sensor de velocidade e atuação é realizada diretamente no acelerador do veículo. O objetivo desse experimento é validar a utilização dos módulos de aquisição e controle, disponibilizados pelo ARCOS.

4.4.2.2 Implementação

Devido ao fato do experimento conter atividades de aquisição de dados e controle, é requerida a especialização dos dois *hot-spots* definidos pelo ARCOS. O experimento foi implementado com a utilização de um modelo matemático que simula o veículo [77], constituindo uma espécie de "planta virtual". O *DAIS Provider* implementado para essa situação, denominado *DAISSimulatedCarProvider*, não utiliza dispositivos reais de aquisição/atuação. Ao invés, ele obtém os dados diretamente do modelo matemático utilizado.

DAIS Simulated Car Provider

O *DAISSimulatedCarProvider* utiliza um modelo matemático para simular o comportamento do veículo ao longo do tempo. Esse modelo é composto por uma função *f(x)*, com a qual pode-se enviar um sinal para o atuador x e verificar a conseqüência desta atuação através da leitura do sensor representado por *f(x)*. As funções abaixo definem o modelo utilizado para simulação do veículo:

$$Accel = \frac{Throttle \times ENGINE\_POWER - (FRIC \times Old\_speed)}{MASS}$$

$$Dist = Old\_speed + Accel \times \left(\frac{1}{SAMPLE\_RATE}\right)$$

$$Speed = \sqrt{Old\_speed^2 + (2 \times Accel \times Dist)}$$

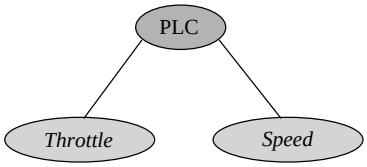


Figura 4.15: Árvore DAIS projetada para o *DAISSimulatedCarProvider*

onde, informados a velocidade atual (*Old_speed*) e o valor da atuação no acelerador (*Throttle*), encontra-se a nova velocidade do veículo. As constantes *ENGINE_POWER* (potência do motor), *FRIC* (coeficiente de fricção), e *MASS* (massa do veículo) são propriedades específicas do veículo sendo modelado, enquanto a constante *SAMPLE_RATE* indica a taxa de amostragem utilizada pelo controlador.

A figura 4.15 apresenta a árvore DAIS projetada para o *DAISSimulatedCarProvider*. Devido à instrumentação simples do experimento, a árvore conta apenas com duas folhas DAIS: uma para representar o acelerador e outra para representar o sensor de velocidade.

O código 4.6 apresenta o arquivo IDL do *DAISSimulatedCarProvider*. Além da convencional disponibilização da faceta *dais_provider* (linha 18), o componente especifica atributos para a configuração das constantes especficas do veículo (linhas 19 a 22).

Código 4.6: Arquivo IDL do componente *DAISSimulatedCarProvider*

```
1  #ifndef ARCOSDAISSIMULATEDCARPROVIDER_IDL
2  #define ARCOSDAISSIMULATEDCARPROVIDER_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
7  #include <ARCOSDAISProviderBase.idl>
8
9  module ARCOS
10 {
11     module DataAcquisition
12     {
13         interface ISimulatedCarProviderFacet : IDAISProviderBaseFacet
14         {
15         };
16         component DAISSimulatedCarProvider
17         {
18             provides ::ARCOS::DataAcquisition::ISimulatedCarProviderFacet dais_provider;
19             attribute float engine_power;
20             attribute float fric;
21             attribute float mass;
22             attribute float sample_rate;
23         };
24         home DAISSimulatedCarProviderHome manages DAISSimulatedCarProvider
25         {
26         };
27     };
28 };
29 #endif /* ARCOSDAISSIMULATEDCARPROVIDER_IDL */
```

PID Controller

O controle da velocidade do carro é realizada, neste experimento, através de um controlador PID discreto, apresentado no código 4.7. Além da convencional disponibilização da faceta *controller* (linha

18), o componente especifica os atributos k_p, k_i e k_d (linhas 19 a 21), utilizados para o *tuning* do controlador e a taxa de aquisição (linha 22). Na implementação do componente *PIDController*, o método *control()* realiza os cálculos PID convencionais.

Código 4.7: Arquivo IDL do componente *PIDController*

```
1  #ifndef ARCOSPIDCONTROLLER_IDL
2  #define ARCOSPIDCONTROLLER_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
7  #include <ARCOSControllerBase.idl>
8
9  module ARCOS
10 {
11     module Control
12     {
13         interface IPIDControllerFacet : IControllerBaseFacet
14         {
15         };
16         component PIDController
17         {
18             provides ::ARCOS::Control::IPIDControllerFacet controller;
19             attribute float kp;
20             attribute float ki;
21             attribute float kd;
22             attribute unsigned short sampling_rate;
23         };
24         home PIDControllerHome manages PIDController
25         {
26         };
27     };
28 };
29 #endif /* ARCOSPIDCONTROLLER_IDL */
```

Supervisão e controle do veículo

Após a especialização dos *hot-spots* e a configuração e implantação da montagem, o controle da velocidade do veículo pode ser acompanhada através do *DAIS Server Browser*. Para que as ações de controle fossem melhor visualizadas, o ARCOS disponibiliza uma aplicação para monitoração gráfica da velocidade do veículo, apresentada na figura 4.16.

Para a utilização dessa aplicação, o usuário deve informar as folhas DAIS que correspondem ao acelerador e ao sensor de velocidade. Para isso, deve-se selecionar a folha DAIS de cada um dos dois dispositivos e clicar no botão correspondente (no grupo "*DAIS Leaves Selection*"). O processo de controle se inicia quando o botão "*Activate/Deactive*" é pressionado. Tanto os parâmetros de configuração do veículo, quanto os parâmetros de *tuning* do controlador, podem ser modificados, sendo esta alteração refletida no processo quando o botão "*Redeploy*" é pressionado. O gráfico apresentado em azul indica a ação de controle realizada na velocidade do veículo. O *slider* à direita da tela permite a mudança da velocidade a ser mantida (*setpoint*).

O *tuning* do controlador foi realizado através do *Método de Ziegler-Nichols* [13, 57]. Esse método estabelece relações empíricas para os valores do ganho proporcional, integral e derivativo, realizando o

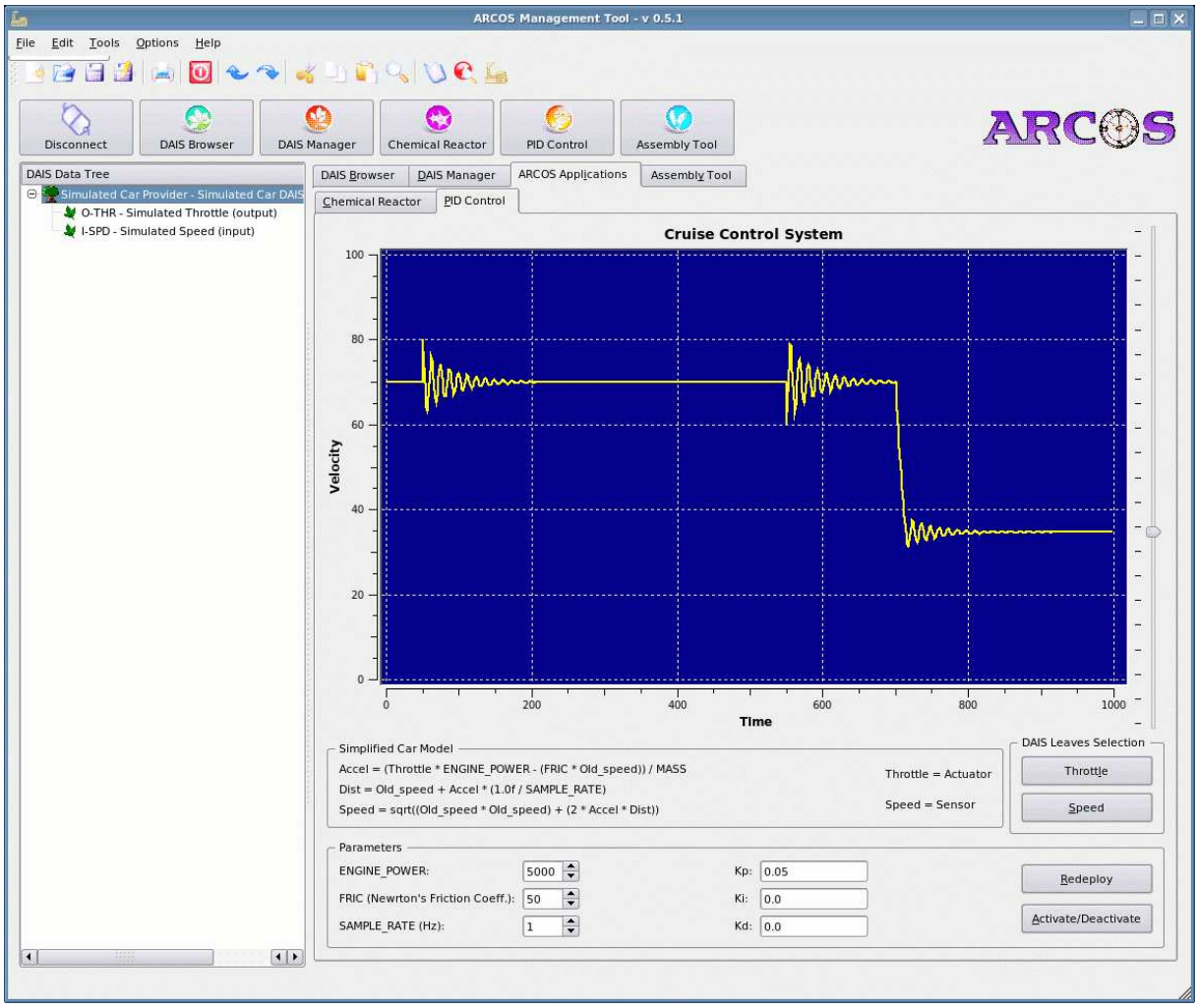


Figura 4.16: Controlador do piloto automático do ARCOS

tuning através da execução do seguintes passos:

1. inicialize os termos integral (k_i) e derivativo (kd) com 0;
2. incremente vagarosamente o termo proporcional (k_p) até que sejam obtidas oscilações contínuas na curva de controle;
3. reduza o termo k_p para a metade do valor encontrado no passo 2;
4. se necessário, atribua valores pequenos para o termo k_d de modo a minimizar os efeitos de *ringing* (estabilização da curva de controle em um valor diferente do *setpoint*).

Este método foi utilizado para *tuning* do controlador do piloto-automático, obtendo os valores: $k_\mathrm{p} = 0.05$ e $k_\mathrm{i} = k_\mathrm{d} = 0$, conforme apresentados na figura 4.16. Variações no *setpoint* (através da movimentação do *slider* à direita da tela) ou nos parâmetros do veículo ou do controlador implicam na realização de uma

reimplantação do controlador PID e do *DAIS Provider*, realizado através do ReDaC.

4.5 Avaliação da plataforma

As sub-seções 4.4.1 e 4.4.2 apresentaram como o ARCOS deve ser utilizado para a construções de novos sistemas industriais de S&C. Com o auxílio do *ARCOS Assembly Tool*, nota-se uma redução considerável no esforço necessário para o desenvolvimento de tais sistemas. Além de verificar as facilidades promovidas pela solução proposta, é necessário a execução de experimentos que avaliem o desempenho e comportamento temporal do ARCOS. Esta seção apresenta resultados de um experimento que visa verificar a execução previsível do Serviço de Eventos, já que este representa o elemento integrador entre os três módulos definidos no ARCOS.

Conforme apresentado na sub-seção 2.4.1.2, o Canal de Eventos de Tempo-Real do TAO contém um módulo de *dispatch* que, em conjunto com o Serviço de Escalonamento, realiza a entrega de mensagens baseada em prioridades. Desta forma, quando dois eventos chegam simultaneamente ao módulo de *dispatch*, será entregue primeiro aquele que obtiver, do Serviço de Escalonamento, uma maior prioridade. Essa prioridade será atribuída em função da estratégia de escalonamento utilizada no Serviço de Escalonamento (atualmente, RMS ou MUF) e dos parâmetros temporais informados ao Serviço de Escalonamento (vide exemplos na seção 2.4.1.2).

Para a verificação desse comportamento, foram criadas instâncias (clientes) do *DAIS Server Browser,* monitorando o experimento do reator químico. Em cada cliente foi criado um grupo de aquisição de dados com diferentes parâmetros temporais. Nota-se que, devido ao fato de grupos de dados se comportarem como produtores de eventos, com o aumento do número de clientes (e conseqüentemente grupos) o Canal de Eventos tende a se sobrecarregar e a dar prioridade para aquelas mensagens destinadas a consumidores de alta prioridade. Essa é uma situação típica na indústria e em sistemas mecatrônicos onde, por exemplo, um sensor precisa enviar informação, ao mesmo tempo, para um controlador (consumidor de alta prioridade) e para um sistema supervisório (consumidor de baixa prioridade).

Os clientes DAIS foram criados a partir da seguinte configuração:

- Cliente c_1 com taxa de aquisição de 5Hz e baixa criticalidade;
- Cliente c_2 com taxa de aquisição de 1Hz e alta criticalidade;
- Clientes c_3 a c_n com taxa de aquisição de 1Hz e baixa criticalidade.

Os parâmetros temporais de cada cliente foram ajustados conforme apresentado nos exemplos da seção 2.4.1.2. Com essa configuração, percebe-se que os clientes c_1 e c_2 serão individualmente beneficiados pelo

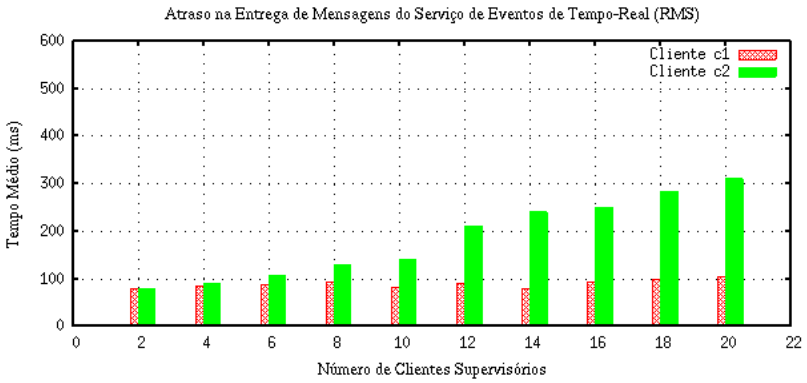


Figura 4.17: Atraso na entrega de mensagens com o escalonamento RMS

uso de uma das duas políticas de escalonamento. Com o Serviço de Escalonamento configurado para uso do algoritmo RMS, terá maior prioridade aquele consumidor com menor período, ou seja, o cliente c_1(período $= 1/5$). Com o Serviço de Escalonamento configurado para uso do algoritmo MUF, o período não será um fator importante e terá maior prioridade aquele consumidor com tiver a maior criticalidade, ou seja, o cliente c_2. Percebe-se que os clientes c_3 a c_n não são beneficiados com nenhuma das duas políticas de escalonamento, pois possuem período menor que o período do cliente c_1 e criticalidade menor que a criticalidade do cliente c_2. Com essa abordagem, quanto maior for o número n de clientes maior será a sobrecarga executada no Canal de Eventos.

As figuras 4.17 e 4.18 apresentam os resultados do experimento com o Serviço de Escalonamento configurado para uso das estratégias RMS e MUF, respectivamente. O Servidor DAIS e os clientes *DAIS Server Browser* foram executados em um *notebook* Pentium Mobile 725, 1.6GHz, *cache* de 2M e 512Mb de memória RAM, ligado ao CLP *Allen-Bradley SLC 5/05* através de uma interface de rede *Ethernet* 10Mbits/s. Em ambos os gráficos, o eixo das ordenadas representa o tempo médio entre a solicitação de entrega do evento (pelo produtor) e sua correspondente recepção (no consumidor), enquanto o eixo das abscissas representa o número total n de clientes DAIS instanciados. Enquanto os clientes c_1 e c_2 têm seus tempos de entrega de mensagens medido, os outros $n-2$ clientes sobrecarregam o Canal de Eventos, tentando perturbar a previsibilidade do sistema.

A figura 4.17 apresenta os resultados do experimento com o Serviço de Escalonamento configurado para uso da estratégia RMS. Com $n = 2$ estão presentes somente os clientes c_1 e c_2 e ambos apresentaram um tempo médio de entrega de 80ms. À medida em que novos clientes DAIS são utilizados, nota-se um aumento substancial do tempo de entrega de mensagens do cliente c_2 (perturbado pelos outros $n-2$ clientes), enquanto o cliente c_1 é priorizado pela estratégia de escalonamento RMS (pelo fato de ter o menor período). Com $n = 4$ (clientes c_1, c_2 e dois clientes perturbadores), o cliente c_2 apresenta um pequeno aumento no atraso (91ms), enquanto o cliente c_1 mantém o atraso por volta dos 80ms. Com $n = 12$ (clientes c_1, c_2 e dez clientes perturbadores), o cliente c_2 apresenta um atraso considerável (210ms), enquanto o cliente c_1 registra um atraso de 90ms. Com $n = 20$ (clientes c_1, c_2 e 18 clientes perturbadores),

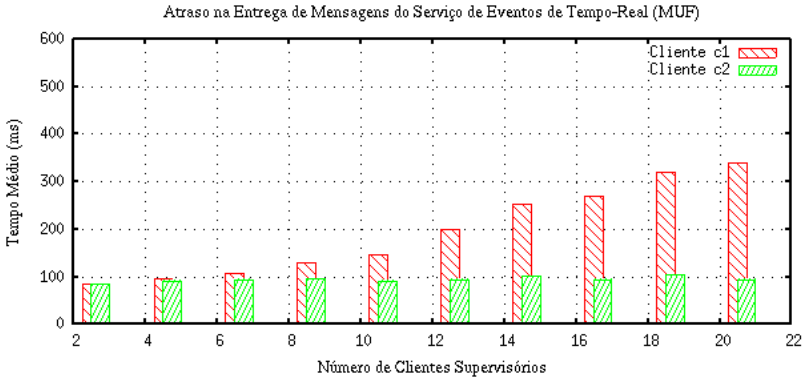


Figura 4.18: Atraso na entrega de mensagens com o escalonamento MUF

o cliente c_2 apresenta um atraso de 310ms, enquanto o cliente c_1 apresenta um atraso de 110ms.

A figura 4.18 apresenta os resultados do experimento com o Serviço de Escalonamento configurado para uso da estratégia MUF. Com $n = 2$ estão presentes somente os clientes c_1 e c_2 e ambos apresentaram um tempo médio de entrega de 80ms. De forma semelhante à execução anterior, novos clientes DAIS foram instanciados e percebe-se, neste caso, a manutenção do tempo de resposta para o cliente c_2, devido ao fato de possuir maior criticalidade e de ser utilizado o escalonamento MUF. Como no exemplo anterior, nota-se a flutuação do tempo de resposta do cliente priorizado (neste caso, o cliente c_2), na faixa dos 100ms.

Os tempos acima descritos foram medidos através de alterações realizadas no componente *DAIS-DAGroupManager*, medindo o tempo decorrido entre a entrega do evento ao Canal de Eventos (após a recepção do *DAIS Provider* devidamente conectado) e a sua recepção no mesmo componente *DAIS-DAGroupManager* (antes do envio ao componente de *callback*), conforme apresentado no diagrama de seqüência da figura 3.6.

Vale ressaltar que o único ponto de execução temporalmente previsível no ARCOS, atualmente, é o Serviço de Eventos de Tempo-Real. Para que toda a plataforma apresente um bom grau de previsibilidade, seriam necessárias garantias na execução dos outros participantes (consumidores e produtores). Uma solução a ser considerada, nesse problema, é a utilização do *container* de tempo-real, previsto para versões futuras do CIAO. Esse *container* hospedará componentes e informará, ao Serviço de Escalonamento, os requisitos temporais demandados por cada um deles.

Apesar de ser atualmente uma plataforma voltada para sistemas críticos em relação ao tempo sem *hard deadlines* [49, 80], o ARCOS apresenta tempo de entrega de mensagens entre 80 e 100ms, adequado para uso em sistemas de controle com carga computacional controlada.

Para que aplicações desenvolvidas sobre o ARCOS possam ser executadas com previsibilidade garantida, o CIAO deve ser executado sobre um sistema operacional de tempo-real, onde serviços disponibilizados pelo *middleware* são mapeados em mecanismos oferecidos pelo sistema operacional. A migração do ARCOS para tais ambientes previsíveis não afeta nenhum dos mecanismos do *framework* aqui proposto,

visto que estes são configurados através da utilização de descritores XML.

4.6 Trabalhos Correlatos

A utilização de componentes distribuídos para o desenvolvimento de sistemas de tempo-real é um tópico de pesquisa recente. Apesar dos constantes estudos sobre aplicações das tecnologias de *middleware* [18, 84], modelos arquiteturais [93], *frameworks* [10, 58], padrões de projeto (*design patterns*) [87], orientação a aspectos [110] e componentes de software [30, 82] na construção de sistemas de tempo-real, alguns poucos resultados práticos foram obtidos ao longo dos últimos anos. Em particular, no caso de componentes distribuídos, a maioria das implementações de modelos padronizados constituem pesquisas em andamento e existem poucos trabalhos validando essas implementações. Portanto, o projeto e implementação de plataformas reutilizáveis e interoperáveis para sistemas de tempo-real ainda constitui um desafio.

Capobianchi et. al. [10] projetaram um *framework,* baseado em CORBA, para a implementação de sistemas industriais de S&C, através da definição de serviços e interfaces para as atividades de aquisição de dados e supervisão. Esse *framework* está definido em três módulos: *Utility*, *Activity* e *Specialized Utility*. O módulo *Utility* provê as funcionalidades básicas de todo sistema de S&C, tais como gerenciamento e acesso de dispositivos de aquisição de dados, definição de eventos, armazenamento de históricos, *logging* de atividades e gerenciamento de segurança (*security*). O módulo *Activity* provê funcionalidades para detecção e notificação de estados críticos, através da definição de situações e anomalias. O módulo *Specialized Utility* provê funcionalidades para sistemas específicos de S&C como, por exemplo, controle de tráfego aéreo ou sistemas de gerenciamento de energia. Além desses serviços, o trabalho propõe uma metodologia, baseada na linguagem formal TRIO, para uso do *framework*. Apesar desses benefícios, aspectos tais como a especificação e garantia de restrições temporais e uso de interfaces padronizadas não foram considerados no trabalho. Além disso, não são apresentadas soluções para malhas fechadas de controle e a utilização do padrão CORBA 2.x implica em uma plataforma menos extensível e flexível.

Em [58], Marti et. al. propuseram um *framework*, baseado em Java, para integração de plantas industriais com o ambiente web. Os autores apresentam o conceito de "planta virtual", formada através da composição de *containers*. Um *container* é uma abstração utilizada para representar um objeto da planta, tal como um sensor, atuador ou um outro *container*. Esses *containers* são organizados em uma árvore e são o mecanismo utilizado para reutilizar a soluções em diferentes configurações de planta. Uma série de objetos Java, implementados com a tecnologia RMI (*Remote Method Invocation*) [62], foram criados para mapear esses *containers* em aplicações web. Além de não considerar soluções para a especificação e garantia das restrições temporais, o trabalho não propõe serviços para implementação de malhas fechadas de controle automático.

Os trabalhos [9] e [50] apresentam propostas de sistemas de S&C baseados em CORBA, porém re-

presentam soluções para situações específicas de sistemas industriais, não considerando a idéia de uma arquitetura reutilizável e flexível.

No trabalho apresentado em [20], Becquet, Locher e Gressier-Soudan propuseram um serviço, baseado em componentes, para a transmissão de mensagens em um ambiente industrial. Essas mensagens, advindas de dispositivos de aquisição, seguem o padrão MMS (*Manufacturing Message Specification*) [15]. Essa pesquisa, apesar de propor um *framework* baseado em componentes (implementado no MicoCCM [29]) e na utilização de interfaces padronizadas (MMS), não apresenta soluções para os aspectos temporais.

Wang et. al. desenvolveram um sistema de tempo-real baseado em componentes que integra a solução de *middleware* com os mecanismos básicos de sistemas operacionais e redes de tempo-real. Além disso, o trabalho também define políticas para controle de admissão de novas tarefas, permitindo modificações *on-line* no conjunto de tarefas. O sistema foi implementado utilizando o sistema operacional de tempo-real *TimeSys* [100] e no JBoss [38], uma implementação livre da especificação *Java 2 Plataform, Enterprise Edition* (J2EE), que contém o padrão EJB para componentes distribuídos. Apesar de representar um importante trabalho para a integração da tecnologia de componentes com os mecanismos básicos dos sistemas operacionais de tempo-real, a pesquisa não está voltada para a definição de plataformas reutilizáveis e interoperáveis.

A tabela 4.2 apresenta uma comparação entre o ARCOS e os trabalhos correlatos aqui apresentados. A coluna "Aquisição, Controle e Supervisão" indica se o trabalho proposto contempla soluções para as três atividades centrais de um sistema industrial de S&C, sendo o ARCOS e o trabalho do Burmakin e Krassi os únicos a apresentarem todas as funcionalidades. No quesito "Integração Web" estão evidenciados o ARCOS (através da versão web do *DAIS Server Browser*) e o *framework* em Java proposto por Marti et. al. A utilização de componentes, e em particular do CCM, no desenvolvimento de sistemas de tempo-real é uma atividade recente e é realizada apenas no ARCOS e no projeto proposto por Becquet, Locher e Gressier-Soudan. Esses dois projetos são também os únicos a apresentarem soluções interoperáveis, alcançadas com o uso do DAIS no ARCOS e do MMS no trabalho do Becquet, Locher e Gressier-Soudan. Apesar de indicações como atividades futuras, os demais trabalhos (com exceção do proposto por Wang et. al.) não apresentam soluções e implementações concretas para suporte à previsibilidade temporal, questão esta tratada no ARCOS através do uso do Serviço de Eventos de Tempo-Real do TAO.

	Aquisição, Controle e Supervisão	Integração Web	Uso de Componentes	Interfaces Padronizadas	Suporte a Tempo-Real
Capobianchi et. al.	Não	Não	Não	Não	Não
Marti et. al.	Não	Sim	Não	Não	Não
Burmakin e Krassi	Sim	Não	Não	Não	Não
Kang, Kim e Park	Não	Não	Não	Não	Não
Becquet, Locher e Gressier-Soudan	Não	Não	Sim	Sim	Não
Wang et. al.	Não	Não	Sim	Não	Sim
ARCOS	Sim	Sim	Sim	Sim	Sim

Tabela 4.2: Comparação do ARCOS com os trabalhos correlatos

Capítulo 5

Conclusões e Trabalhos Futuros

Esta dissertação apresentou uma solução flexível, reutilizável e interoperável para o desenvolvimento facilitado de sistemas industriais de S&C baseados em componentes. O projeto e desenvolvimento dessas plataformas constitui uma atividade fundamental para o atendimento dos requisitos dos sistemas de tempo-real modernos. Este capítulo apresenta as conclusões sobre o trabalho proposto, as perspectivas futuras para utilização e extensão do ARCOS e as publicações resultantes deste projeto.

5.1 Conclusões

Os sistemas de tempo-real atualmente em produção são geralmente caracterizados pelo uso de uma plataforma de software e hardware específicos para a situação em questão, consistindo soluções inflexíveis, com baixo grau de reutilização e com dificuldades de comunicação com outros sistemas. Além disso, as demandas atuais por funcionalidades envolvendo distribuição, flexibilidade, extensibilidade, adaptação, uso de algoritmos inteligentes, interoperabilidade e reutilização exigem uma infra-estrutura computacional (hardware e software) não disponibilizada pelas soluções baseadas em CLP's e programação em LADDER. No desenvolvimento dessa classe de sistemas, deseja-se aliar alta produtividade e alta qualidade com uma redução no custo e tempo necessários para o desenvolvimento de novas aplicações.

Uma tendência, considerada nos últimos anos, é a utilização de componentes COTS de hardware e software. Além de uma considerável redução nos custos, as facilidades para interoperabilidade e integração com outros ambientes é substancialmente alavancada. Como exemplo, pode-se citar os sistemas de controles baseados em PC's [35], que tentam aliar o baixo custo e interoperabilidade dos PC's com a robustez e previsibilidade dos CLP's. Com essa melhor infra-estrutura de hardware e software, metodologias e paradigmas para a gerência da complexidade, criada pelas demandas atuais, se tornam passíveis de utilização. Tecnologias já consolidadas no desenvolvimento de Sistemas de Informação, tais como a

orientação a objetos, padrões de projeto (*design patterns*), *middleware*, *frameworks*, componentes distribuídos e orientação a aspectos, representam soluções para problemas que começam a surgir no domínio dos sistemas industriais de tempo-real. O desafio é utilizar essas tecnologias mantendo, ao mesmo tempo, a disponibilidade, robustez e previsibilidade requeridos por tais sistemas.

Esta dissertação apresentou o ARCOS (ARquitetura para COntrole e Supervisão), um *framework* vertical, do tipo caixa branca, para desenvolvimento de sistemas distribuídos de S&C baseados em componentes de tempo-real. Além disso, um grupo de ferramentas, agrupadas no *ARCOS Management Tool*, torna possível a utilização imediata do ARCOS nas principais atividades de supervisão e controle de uma planta industrial. As principais metas do *framework* proposto são: proporcionar um bom grau de reutilização e uma conseqüente melhoria da produtividade, interoperabilidade e flexibilidade dos sistemas industriais de S&C.

O capítulo 4 apresentou os passos necessários para especializar o ARCOS a uma situação específica de S&C. Com a realização dos cinco passos descritos, uma nova aplicação de S&C pode ser construída com um esforço consideravelmente reduzido. Ao utilizar o *ARCOS Assembly Tool*, apresentado na seção 4.3, essa atividade se torna ainda mais fácil, podendo ser realizada por pessoas sem um conhecimento aprofundado em CCM e CIAO.

Ao adotar o padrão DAIS (*Data Acquisition from Industrial Systems*), o ARCOS disponibiliza uma solução interoperável para aquisição e alteração de dados industriais. Pelo fato de ser um padrão CORBA, a interoperabilidade adquirida atravessa barreiras relacionadas com linguagens de programação, sistemas operacionais, arquiteturas de hardware e meios de comunicação. A versão web do *DAIS Server Browser*, disponibilizada pelo ARCOS, representa uma resposta para a forte demanda de integração dos sistemas de S&C com o ambiente web. A integração com outros padrões de aquisição de dados, tais como o OPC (*OLE for Process Control*) e o MMS (*Manufacturing Message Specification*), pode ser alcançada através da implementação de *DAIS Providers* que realizam o interfaceamento entre os dois padrões em questão.

Com a utilização da tecnologia de componentes, e em particular o CCM, obtém-se uma arquitetura de software que apresenta características que favorecem a flexibilidade, reutilização e a escalabilidade em sistemas industriais de S&C. Ao desenvolver um *framework* utilizando a tecnologia de componentes, a especialização dos *hot-spots* acontece através de conexões entre os componentes definidos pela arquitetura e os componentes implementados pelo usuário. Em particular, a especificação para implantação e configuração de componentes [72], criada pelo OMG e implementada no CIAO como o DAnCE (*Deployment And Configuration Engine*), apresenta facilidades para utilização de repositórios de componentes e para a implantação remota de componentes em diversos nós do dóminio de implantação. O ReDaC (*Redeployment and Reconfiguration*) permite que uma montagem (configuração de conexões entre componentes) seja alterada, sem a interrupção do sistema. Ao fazer uso de todas essas tecnologias, o ARCOS se beneficia dos aspectos acima citados, ao mesmo tempo em que define uma arquitetura adequada para

a implementação das principais atividades presentes em sistemas industriais de S&C.

Grande parte dos sistemas industriais de controle é caracterizada pela necessidade de um comportamento temporalmente previsível. Em sistemas de tempo-real, essa previsibilidade é alcançada através da utilização conjunta de uma série de tecnologias, tais como: sistemas operacionais de tempo-real, *middleware* e linguagens de programação para tempo-real e redes de comunicação em tempo-real. O ACE e o TAO, tecnologias subjacentes na arquitetura do ARCOS, são soluções já consolidadas no desenvolvimento de sistemas de tempo-real distribuídos, tais como sistemas de controle de tráfego aéreo e sistemas de comutação telefônica. O CIAO, pelo fato de ser um trabalho recente, ainda não foi aplicado na construção de sistemas industriais em produção, porém o seu objetivo mantém a proposta de aplicação em sistemas de tempo-real. O ARCOS faz uso extensivo do Serviço de Eventos de Tempo-Real do TAO para realizar a entrega de mensagens baseada em prioridades. Futuramente, quando o CIAO disponibilizar os demais mecanismos para tempo-real (como, por exemplo, o *container* de tempo-real), o ARCOS contará com um comportamento temporal cada vez mais previsível.

O projeto ARCOS é composto pelos seguintes participantes: o *framework* reutilizável (componentes *server-side*), aplicações-cliente genéricas (*DAIS Server Browser* - *desktop* e web - e o *DAIS Server Manager*), ferramentas de auxílio ao desenvolvimento de aplicações baseadas no ARCOS (*ARCOS Assembly Tool*) e aplicações-exemplo (supervisão do reator químico e controle PID para piloto automático). A utilização conjunta dessas tecnologias faz do ARCOS uma boa alternativa na escolha de plataformas flexíveis e reconfiguráveis para o desenvolvimento de sistemas industriais modernos.

Dentre as contribuições do trabalho apresentado nesta dissertação, pode-se destacar: validação de uma implementação do padrão CCM, aglutinação de *expertise* acerca dos sistemas industriais de S&C em uma plataforma flexível, interoperável e reutilizável e a disponibilização de uma arquitetura para validação de novas tecnologias de aquisição de dados e controle.

Devido ao fato de ser um padrão recente, o CCM, cuja primeira versão foi lançada em junho de 2002, não conta com uma grande variedade de implementações e os esforços atuais constituem trabalhos em andamentos, com implementações parciais da especificação. Desse modo, a existência de projetos utilizando e validando essas implementações é uma atividade fundamental para que, através de relatos de problemas e resultados alcançados, estas soluções alcancem o nível de maturidade exigido para aplicação em sistemas reais. O ARCOS, no momento da escrita deste texto, representa o primeiro esforço (conforme apresentado em *http://www.cs.wustl.edu/~schmidt/TAO-users.html*) de uso extensivo das tecnologias presentes no CIAO, tais como o DAnCE e o ReDaC, contribuindo com relatos de *bugs*, contribuições de códigos-fonte e sugestões para a melhoria desse *middleware*.

Ao projetar a plataforma sob o formato de um *framework*, conhecimentos e soluções adequadas para a implementação de sistemas industriais de S&C estão inerentemente aglutinados. Ao utilizar as inversões de controle presentes no ARCOS, o desenvolvedor faz uso de uma solução arquitetural adequada e validada

para o domínio de aplicação em questão, reduzindo as possibilidades de projeto de arquiteturas inflexíveis e/ou que não atendem os requisitos demandados por tais sistemas.

A definição dos *hot-spots* para aquisição de dados e controle faz do ARCOS uma plataforma interessante para a validação de novas tecnologias. A utilização de dispositivos *wireless* ou comandados por portas USB é facilmente alcançada no ARCOS através da implementação de novos *DAIS Providers*, enquanto todo o arcabouço necessário para integração e supervisão dessa informação já está implementada de forma reutilizável. O projeto, implementação e validação de estratégias modernas de controle, tais como controle preditivo, multi-variável ou adaptativo, pode ser alcançado no ARCOS através da implementação de novos controladores, os quais serão facilmente conectados à estrutura definida pelo *framework* proposto.

5.2 Trabalhos futuros

A arquitetura modular e flexível apresentada no ARCOS possibilita a sua extensão de modo a acomodar outros requisitos funcionais e não-funcionais. Dentre os possíveis trabalhos futuros, pode-se destacar: avaliação do ARCOS em um ambiente de sistema operacional e redes de tempo-real, experimentos do ARCOS com sistemas de controle reais (não-simulados), melhorias nas ferramentas de auxílio ao desenvolvimento, extensões para *dependability* e extensões para comportamento auto-adaptativo.

Enquanto este trabalho aplicou técnicas de engenharia de software para a construção de uma solução reutilizável e interoperável no domínio dos sistemas industrais de S&C, é de fundamental importância a avaliação da plataforma proposta em ambientes caracterizados pela utilização de um sistema operacional de tempo-real e de uma rede de comunicação de tempo-real. Apesar da utilização do Serviço de Escalonamento e da execução do módulo de *dispatch* do Canal de Eventos como um escalonador de alta prioridade, a utilização de um sistema operacional de tempo-real proporciona a utilização de outras políticas, tais como a EDF (*Earliest Deadline First*) [80], bem como soluções para problemas clássicos como, por exemplo, o problema da inversão de prioridades [80].

Um trabalho futuro importante para a validação do comportamento temporal do ARCOS é a sua utilização na implementação de sistemas reais de controle, com requisitos estritos de tempo. Nesses experimentos será possível verificar qual a influência que a plataforma de software irá exercer na manutenção de um controle estável. Como exemplos desses experimentos pode-se citar: o pêndulo invertido, controle de velocidade de carros em autoramas, o levitador magnético, dentre outros. A utilização do ARCOS nesses experimentos requer a sua implantação em um ambiente de tempo-real, conforme apresentado no parágrafo anterior.

O ARCOS disponibiliza soluções para as principais atividades de supervisão: verificação dos dados disponibilizados pelo servidor (árvore DAIS), seleção de dados a serem adquiridos numa determinada

freqüência e monitoração *on-line* da aquisição realizada. Além disso, o *ARCOS Assembly Tool* representa uma ferramenta importante para a construção de novas aplicações baseadas no ARCOS. Como trabalhos futuros na expansão dessas ferramentas pode-se citar: definição de mecanismos para segurança (controle de acesso), registro de atividades e *logging* no *DAIS Server Browser*, implementação de um sistema para construção de supervisórios específicos, melhorias no *ARCOS Assembly Tool* para a criação de *profiles* de implantação e consulta de repositório de componentes.

Um requisito freqüentemente implementado em sistemas supervisórios é a definição de políticas de controle de acesso às funcionalidades disponibilizadas. Por exemplo, alguns usuários devem poder realizar somente a monitoração do sistema, enquanto outros serão capazes de alterar valores de *setpoint*. Além disso, é interessante manter um registro das atividades realizadas pelos usuários do sistema supervisório, de modo a identificar operações passadas com objetivos de auditoria ou verificação de causas de um determinado problema. Para fins de armazenamento histórico da planta, é comum a utilização de sistemas gerenciadores de banco de dados para armazenar informações do chão-de-fábrica, possibilitando a extração de relatórios e estimativas gerenciais. Essas funcionalidades não estão atualmente implementadas no *ARCOS Management Tool* e constituem um trabalho futuro.

A existência de uma ferramenta que auxilie a construção de supervisórios específicos é também fator importante para o sucesso destas aplicações. Apesar do supervisório específico do reator químico ter sido construído através de atividades de programação, uma ferramenta que possibilitasse a construção de cenas representando a planta e seu eventos é altamente desejável. Através dessa ferramenta, o chão-de-fábrica seria representado pela inclusão de figuras pré-definidos (válvulas, tanques, motores, sensores) ou criadas pelo usuário. Eventos no chão-de-fábrica, tais como aberturas de válvulas ou acionamento de sensores, seriam representados por mudanças de cores ou animações nas figuras. A configuração das figuras e dos eventos seria realizada, pelo usuário, na ferramenta em questão, dispensando atividades de programação. A ferramenta para construção facilitada de supervisórios específicos também é um trabalho futuro no projeto ARCOS.

Grande parte dos sistemas de tempo-real, devido ao fato de estar associado a tarefas críticas, necessita de mecanismos para obtenção de um certo grau de *dependability*[1]. A implementação de soluções para detectores de defeitos, bem como o gerenciamento (ativo, passivo ou semi-ativo) de réplicas é uma atividade futura prevista no projeto do ARCOS. Iniciativas de soluções, baseadas no CORBA, para Tolerância a Falhas, tais como o FT-CORBA [74, 70] e as implementações realizadas pelo TAO, formam uma infra-estrutura bastante apropriada para utilização no projeto do ARCOS.

Uma característica bastante desejada, atualmente, no sistemas de tempo-real é a possibilidade de adaptação dinâmica como resposta a mudanças no ambiente operacional. Essa adaptação é justificada ou pela inexistência de uma solução adequada para todos os casos ou pela impossibilidade de prever

[1] Qualidade do serviço computacional disponibilizado, de modo que a confiança no funcionamento é uma característica justificada [5, 56].

todas as situações ambientais às quais o sistema será submetido. A possibilidade de uso de um conjunto de soluções, onde uma delas é escolhida para uso em uma determinada situação, é uma funcionalidade importante. Quando a necessidade de uma troca de solução (adaptação) é percebida e realizada, em tempo de execução, pelo próprio sistema, este é dito um sistema auto-adaptativo e adaptação é dita dinâmica [76]. Para alcançar esse comportamento, a aplicação deve apresentar uma estrutura flexível, consciente das possibilidades de adaptação e com mecanismos para realização de mudanças em tempo de execução, sem a interrupção do sistema. Em particular, com o uso do CIAO, esse comportamento pode ser implementado através da: *i*) definição de uma arquitetura para armazenamento da estrutura interna da montagem (meta-modelo), *ii*) implementação de componentes para políticas de adaptação (critérios que definem quando uma adaptação é necessária) e *iii*) uso do ReDaC para a realização de mudanças nas instâncias, conexões e valores de atributos presentes na montagem. O CIAO, DAnCE e ReDaC já apresentam todas as funcionalidades necessárias para a implementação de comportamento adaptivo e este é um trabalho futuro no projeto do ARCOS.

Alguns dos trabalhos futuros acima citados constituem partes de dissertações atualmente em andamento no Laboratório de Sistemas Distribuídos (LaSiD), no âmbito do Programa de Pós-Graduação em Mecatrônica (PPGM) da UFBa.

5.3 Publicações

A elaboração deste projeto de pesquisa resultou na aprovação dos seguintes artigos científicos (listados em ordem de importância):

- *A Component-Based Real-Time Architecture for Distributed Supervision and Control Applications.*

  - Autores: Sandro Santos Andrade e Raimundo José de Araújo Macêdo.
  - Veículo de publicação: *Proceddings of ETFA 2005 - 10th IEEE International Conference on Emerging Technologies and Factory Automation. Catania, Italy. New York: IEEE Computer Society Press. Pages 15-22.*
  - Resumo: *Nowadays, the development of flexible and interoperable software platforms for industry is an important issue. The CCM model captures two paramount features of such platforms, as it combines component-based middleware (easy composition of new applications and maintainability) and the openness of the CORBA standard. Though there exist an implementation of CCM devoted to real-time systems (CIAO platform), much effort is needed to validate its use in the real-time industry scenario. This paper contributes to this goal by presenting the design*

and implementation of a new framework over CIAO, which conforms to the DAIS standard (Data Acquisition from Industrial Systems). We discuss our design decisions and show how the framework can be used to develop distinct S|&C applications. We also discuss implementation details and show performance data from a series of experiments.

- *Using Real-Time Components to Construct Supervision and Control Applications.*

  - Autores: Sandro Santos Andrade and Raimundo José de Araújo Macêdo.
  - Veículo de publicação: *8th Brazilian Workshop on Real-Time Systems - Work-in-Progress Paper. June 2, 2006. Curitiba - PR, Brazil.*
  - Resumo: *Integration and interoperability are major challenges of modern supervision and control industrial systems. The ARCOS platform is being developed to address this important issue of modern industrial systems. It is based on the CCM model, which combines component-based middleware (easy composition of new applications and maintainability) and the openness of the CORBA standard. This paper overviews the current development stage of the ARCOS platform, and presents preliminary results originated from the development of a few supervision and control applications.*

- ARCOS: A Component-Based Architecture for the Construction of Robust Supervision and Control Applications.

  - Autores: Sandro Santos Andrade, Raimundo José de Araújo Macêdo and Alírio Sá.
  - Veículo de publicação: *Workshop on Dependable Automation Systems. 2nd. Latin-American Symposium on Dependable Computing. October 25-28, 2005. Salvador - BA, Brazil.*
  - Resumo: *Integration and interoperability are major challenges of modern supervision and control industrial systems. Such needs arise from the use of equipments from different vendors, with operating systems and communications incompatibilities, up to diverse factory machinery - each of them usually supplied by a specialized company (PLC's, numerical control machines, robot arms). The ARCOS platform has been developed to address this important issue of modern industrial systems. It is based on the CCM model, which combines component-based middleware (easy composition of new applications and maintainability) and the openness of the CORBA standard. This paper overviews the ARCOS platform and discusses its components*

designed to provide a flexible failure detection mechanism. Such components can be customized to handle distinct QoS requirements and be adaptive to the current system and network load.

- *Real-Time Component Software for Flexible and Interoperable Automation Systems.*

  - Autores: Sandro Santos Andrade and Raimundo José de Araújo Macêdo.
  - Veículo de publicação: XVI Congresso Brasileiro de Automática. 3 a 6 de outubro de 2006. Salvador - BA, Brasil.
  - Resumo: *The use of software-intensive solutions for industrialautomation and control systems has been a promising trend due to the flexibility, interoperability, and cost savings provided by such an approach. This paper presents a component-based framework for the development of industrial supervision and control systems, which provides reusable solutions for data acquisition, control, and supervision activities. Interoperability and real-time issues are handled by the implementation of the DAIS (Data Acquisition for Industrial Systems) standard, in conjunction with the use of CIAO, a real-time implementation of the CCM (CORBA Component Model) standard. We present the design and implementation of the proposed framework, as well as two application examples built atop our software platform: a chemical reactor supervisory and a cruise control system simulation.*

- *A Management Tool for Component-Based Real-Time Supervision and Control Systems.*

  - Autores: Sandro Santos Andrade and Raimundo José de Araújo Macêdo.
  - Veículo de publicação: 13a. Sessão de Ferramentas. XX Simpósio Brasileiro de Engenharia de Software. 16 a 20 de outubro de 2006. Florianópolis - SC, Brasil.
  - Resumo: *In the last years, many researches have been concerned with the adaptation of software engineering techniques for use in real-time industrial systems. Following this context, we have designed and implemented the ARCOS framework, devoted to the construction of reusable, flexible, and interoperable industrial systems. This paper presents a management tool we have developed to facilitate the specialization of ARCOS into a particular industrial scenario and that provides a generic client for real-time supervision and control of industrial plants. We briefly discuss the ARCOS server-side component architecture, present two application examples, and describe the provided facilities for the configuration, assembly, and deployment of ARCOS-based applications.*

- Tratando a Previsibilidade em Sistemas de Tempo-Real Distribuídos: Especificação, Linguagens, Middleware e Mecanismos Básicos (in portuguese).

  - Autores: R. J. A. Macêdo, G. M. Lima, L. P. Barreto, A. M. S. Andrade, F. J. R. Barboza, A. Sá, R. Albuquerque, and S. S. Andrade.
  - Veículo de publicação: Capítulo 3 do livro texto do mini-curso apresentado no XXII Simpósio Brasileiro de Redes de Computadores, SBRC 2004. 10 a 14 de maio de 2005. Gramado - RS, Brasil. Páginas 105-163. ISBN: 85-88442-82-5.
  - Comentários: Capítulo de livro do mini-curso apresentado no XXII Simpósio Brasileiro de Redes de Computadores. Apesar de não ser um trabalho que apresenta diretamente o ARCOS, possui seções dedicadas às soluções de *middleware* para tempo-real, em particular, o TAO.
  - Resumo: Garantir a previsibilidade de um sistema de tempo-real distribuído envolve uma série de técnicas complementares, que vão desde a especificação e verificação formal do sistema, passando pelo uso de middleware e linguagens de programação, até o gerenciamento de recursos que garantam um comportamento previsível do ambiente computacional (hardware e software). O objetivo deste capitulo é dar ao leitor uma visão integrada dos diversos aspectos de projeto de sistemas de tempo-real, mostrando como esses aspectos se inter-relacionam e interferem no processo de desenvolvimento de tais sistemas.

Para maiores informações sobre o projeto ARCOS, consultar o site http://arcos.sourceforge.net.

Referências Bibliográficas

[1] *OpenCCM - The Open CORBA Component Model Platform.* http://openccm.objectweb.org.

[2] J. Hansson A. Tesanovic, D. Nystrom and C. Norstom. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, February 2004.

[3] Adam Childs et al. Cadena: An Integrated Development Environment for Analysis, Synthesis, and Verification of Component-Based Systems. In *FASE 04*, volume 2984 of *Lecture Notes in Computer Science*, pages 160–164. Springer, 2004.

[4] D. M. Auslander. What is mechatronics ? *IEEE/ASME Transactions on Mechatronics*, 1(1):5–9, 1996.

[5] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability, 2001.

[6] David E Bakken. Middleware. *Washington State University. Encyclopedia of Distributed Computing, Kluwer Academic Press, 2003.*, 2003.

[7] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.

[8] U. Brinkschult and Th. Ungerer. A microkernel middleware architecture for distributed embedded real-time systems. *20th IEEE Symposium on Reliable Distributed Systems (SRDS'01)*, October 2001.

[9] E. M. Burmakin and B.A. Krassi. Distributed automation and control systems. *Preprints of the 9th International Student Olympiad on Automatic Control (Baltic Olympiad).*

[10] Riccardo Capobianchi, Alberto Coen-Porisini, Dino Mandrioli, and Angelo Morzenti. A framework architecture for supervision and control systems. *ACM Comput. Surv.*, 32(1es):26, 2000.

[11] J.E. Carryer. The design of laboratory experiments and projects for mechatronics courses. *Workshop on Mechatronic Curriculum Development*, 1996.

[12] Rômulo Silva de Oliveira e Carlos Montez Cassia Yuri Tatibana. Um estudo das propostas de modelos de componentes para sistemas de tempo real. *WTR'2004 - 6o Workshop de Tempo Real. Gramado - RS, 14 de maio de 2004*, 2004.

[13] Control Engineering Resource Center. *Control Engineering Reference Guide to PID Tuning (Part 1).* http://resource.controleng.com/community/23094/Library.html.

[14] Douglas C. Schmidt Christopher D. Gill and Ron Cytron. Multi-paradigm scheduling for distributed real-time embedded computing. *IEEE Proceedings Special Issue on Modeling and Design of Embedded Systems*, 91(1), 2003.

[15] L. Ciminiera, C. Demartini, and A. Valenzano. *MAP and TOP Communications: Standards and Applications.* 1992.

[16] Douglas E. Comer. *Internetworking with TCPIP Vol I: Principles, Protocols and Architecture. 2nd ed.* Prentice Hall, 1991. COM d 91:1.

[17] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.

[18] Manuel Diaz and Daniel Garrido. Applying RT-CORBA in nuclear power plant simulators. In *ISORC*, pages 7–14, 2004.

[19] Hans Rohnert Douglas C. Schmidt, Michael Stal and Frank Buschmann. Pattern-oriented software architecture: Patterns for concurrent and networked objects. *Wiley & Sons*, 2000.

[20] E. Gressier-Soudan E. Becquet, H-N. Locher. Component-based industrial messaging service design for utilities. *ETFA'03. 9th IEEE International Conference on Emerging Technologies and Factory Automation*, 2003.

[21] Will Tracz (editor). Tutorial: Software reuse: Emerging technology. *IEEE Computer Society Press*, 1988.

[22] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional; 1st edition, 1995.

[23] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. Building application frameworks: object-oriented foundations of framework design. *John Wiley & Sons, Inc.*, 1999.

[24] Blender Foundation. *Blender: Open-Source 3D Graphics Creation.* http://www.blender.org.

[25] Free Software Foundation. *GNU Automake.* http://www.gnu.org/software/automake.

[26] Free Software Foundation. *GNU Make.* http://www.gnu.org/software/make.

[27] Free Software Foundation. *The GNU Operating system - the GNU project - Free Software Foundation - Free as in Freedom - GNU/Linux.* http://www.gnu.org/.

[28] OPC Foundation. *OLE for Process and Control Standard.* http://www.opcfoundation.org, 1997.

[29] FPX. *The MICO CORBA Component Project.* http://www.fpx.de/MicoCCM/.

[30] Johan Fredriksson, Mikael Åkerholm, Kristian Sandström, and Radu Dobrin. Attaining flexible real-time systems by bringing together component technologies and real-time systems theory. In *EUROMICRO*, pages 399–403, 2003.

[31] W. Otte D. Schmidt G. Deng, J. Balasubramanian and A. Gokhale. DAnCE: A qos-enabled component deployment and conguration engine. *Proceedings of the 3rd Working Conference on Component Deployment, Grenoble, France, November 28-29*, 2005.

[32] Ron Gage. *Allen-Bradley Ethernet Library (v. 0.1.8).* http://abplc5.sourceforge.net/, 2005.

[33] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time CORBA scheduling service. *Real-Time Systems*, 20(2):117–154, 2001.

[34] Itana Maria De Souza Gimenes and Elisa Hatsue Moriya Huzita. Desenvolvimento baseado em componentes: Conceitos e técnicas. *Ciencia Moderna*, 2005.

[35] Mikell P. Groover. Automation, production systems, and computer-integrated manufacturing. *Prentice Hall*, 2001.

[36] Debian Group. *Debian - The Universal Operating System.* http://www.debian.org/.

[37] DOC Group. *Distributed Object Computing (DOC) Group for DRE Systems.* http://www.dre.vanderbilt.edu/.

[38] JBoss Group. *JBoss.ORG.* http://www.jboss.org/.

[39] KDE Group. *K Desktop Environment.* http://www.kde.org/.

[40] KDevelop Group. *KDevelop - um Ambiente Integrado de Desenvolvimento (IDE).* http://www.kdevelop.org/.

[41] Object Management Group. *CORBA Specification.* http://www.omg.org/gettingstarted/corbafaq.htm.

[42] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of OOPSLA '97*, pages 184–200, 1997.

[43] Bonnie S. Heck. *Software Technologies for Complex Control Systems.* http://users.ece.gatech.edu/bonnie/DARPA/Software_Technologies_tutorial.pdf, 2001.

[44] Bonnie S. Heck, Linda M. Wills, and George J. Vachtsevanos. Software enabled control: Background and motivation. In *American Control Conference*, June 2002.

[45] George T. Heineman and William T. Councill. Component based software engineering: Putting the pieces together. *Addison-Wesley Professional; 1st edition*, 2001.

[46] E. M. Hemerly. *Controle por computador de sistemas dinâmicos.* Edgard Blücher, 1996.

[47] J. Hu and D. Schmidt. JAWS: A framework for high performance web servers. *Domain-Specific Application Frameworks: Frameworks Experience by Industry. Wiley & Sons*, 1999.

[48] M. B. Dwyer G. Jung J. Hatcliff, W. Deng and V. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. *International Conference on Software Engineering - ICSE 2003*, 2003.

[49] Rômulo Silva de Oliveira Jean-Marie Farines, Joni da Silva Fraga. Sistemas de tempo real. *Universidade Federal de Santa Catarina, Departamento de Automação e Sistemas.*

[50] Weonjoon Kang, Hyoungyuk Kim, and Hong Seong. Park. Design and performance analysis of middleware-based distributed control systems. *Kangwon National University. IEEE.*, 2001.

[51] K.H. (Kane) Kim. An efficient middleware architecture supporting time-triggered message-triggered objects and an NT-based implementation. *Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 1999.

[52] K.H. (Kane) Kim. APIs for real-time distributed object programming. *IEEE Computer*, pages pp, 72–80, June 2000.

[53] K.H. (Kane) Kim. Object-oriented real-time distributed programming and support middleware. *Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*, July 2000.

[54] Mary Kirtland. Projetando soluções baseadas em componentes. *Editora Campus*, 2000.

[55] Hermann Kopetz. Real-time systems : Design principles for distributed embedded applications. *Springer; 1 edition*, 1997.

[56] J. C. Laprie. Dependable computing and fault tolerance: concepts and terminology. pages 2–11, June 1985.

[57] A. F. Almeida M. Embiruçu, C. de F. A. Neto. *Controle PID.* Apostila de Curso; Apoio a Cursos de Extensão, Graduação e Pós-Graduação - DEQ/UFBa, 2003.

[58] P. Marti, J.C. Aguado, F. Rolando, M. Velasco, J. Colomar, and J.M. Fuertes. A java-based framework for distributed supervision and control of industrial processes. *7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA99)*, 1:33–41, 1999.

[59] Steve Vinoski Michi Henning. *Advanced CORBA Programming with C++.* Addison-Wesley Professional; 1st edition (February 17, 1999), 1999.

[60] Microsoft. *COM: Component Object Model Technologies.* http://www.microsoft.com/com/default.mspx.

[61] Sun Microsystems. *EJB 2.1 Specification.* http://java.sun.com/products/ejb/docs.html.

[62] Sun Microsystems. RMI: Remote method innvocation. http://java.sun.com/rmi.

[63] Sun Microsystems. RPC: Remote procedure call protocol specification. version 2. *Req. For Com. 1957, Sun Microsystems, Inc.*, June 1988.

[64] MinGW. *MinGW.* http://www.mingw.org/.

[65] D. C. Schmidt N. Wang, C. Gill and V. Subramonian. Configuring real-time aspects in component middleware. *International Symposium on Distributed Objects and Applications - DOA 2004*, 2004.

[66] Inc. OCI Object Computing. *The Makefile, Project, and Workspace Creator (MPC).* http://download.ociweb.com/MPC/MakeProjectCreator.pdf.

[67] K. Ogata. *Engenharia de Controle Moderno.* Prentice-Hall do Brasil; 2a. edição, 1993.

[68] Object Management Group (OMG). *Object Management Group.* http://www.omg.org.

[69] Object Management Group (OMG). *Data Acquisition from Industrial Systems (DAIS), Request for Proposal (RFP), OMG Document: dtc/99-01-02.* http://www.omg.org/techprocess/meetings/schedule/Data_Acquisition_RFP.html, 1999.

[70] Object Management Group OMG. *Fault Tolerant CORBA Specification, V1.0, OMG Document ptc/2000-12-06 ed., April 2000, OMG Final Adopted Specification.* 2000.

[71] Object Management Group (OMG). *CORBA Component Model.* http://www.omg.org/technology/documents/formal/components.htm, 2001.

[72] Object Management Group (OMG). *Deployment and Configuration Adopted Submission.* OMG Document ptc/03-07-08 edn, 2003.

[73] Object Management Group (OMG). *Real-Time CORBA.* http://www.omg.org/technology/documents/formal/RT_dynamic.htm, 2003.

[74] Object Management Group (OMG). *Fault-Tolerant CORBA.* Capítulo 23 de http://www.omg.org/technology/documents/formal/corba_iiop.htm, 2004.

[75] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[76] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software, 1999.

[77] Michael J. Pont. Patterns for time-triggered embedded systems. *Addison-Wesley Professional; 1st edition (July 12, 2001)*, 2001.

[78] Richard Schantz Jianming Ye Prakash Manghwani Matthew Gillen Praveen Sharma, Joesph Loyall and George T. Heineman. Using composition of qos components to provide dynamic, end-to-end qos in distributed embedded applications - a middleware approach. *IEEE Internet Computing for Application-Level QoS*, 2006.

[79] Arno Puder and Kay Römer. MICO – MICO is CORBA. *dpunkt-Verlag*, 1998.

[80] L. P. Barreto A.M.S.Andrade F.J.R.Barboza A. Sá R. Albuquerque S. Andrade R. J. A Macêdo, G. M. Lima. Tratando a previsibilidade em sistemas de tempo-real distribuídos: Especificação. linguagens, middleware e mecanismos básicos. *Capítulo 3 do Livro texto para o mini-curso do 22o. Simpósio Brasileiro de Redes de Computadores, SBRC'2004, pp. 105-163, ISBN: 85-88442-82-5, 10 a 14 de maio de 2004, Gramado, RS.*

[81] Gregory F. Rogers. Framework-based software development in c++. *Prentice-Hall, Inc.*, 1997.

[82] Wendy Roll. Towards model-based and CCM-based applications for real-time systems. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 75. IEEE Computer Society, 2003.

[83] Johannes Sametinger. Software engineering with reusable components. *Springer; 1 edition*, 2001.

[84] Ricardo Sanz. A CORBA-based architecture for strategic process control. *Annual Reviews in Control*, 27(1):15–22, 2003.

[85] Ricardo Sanz and Mariano Alonso. CORBA for control systems. *Annual Reviews in Control*, 25(1):169–181, 2001.

[86] Ricardo Sanz, José Antonio Clavijo, Angel de Antonio, and Miguel Segarra. ICa: Middleware for intelligent control. *International Symposium on Intelligent Control, Intelligent Systems, and Semiotics*, September 1999.

[87] Ricardo Sanz and Janusz Zalewski. Pattern-based control systems engineering. *IEEE Control Systems*, 23(3):43–60, July 2003.

[88] Douglas C. Schmidt. *Who is using ACE/TAO/CIAO.* http://www.cs.wustl.edu/~schmidt/TAO-users.html.

[89] Douglas C. Schmidt. GPERF: A perfect hash function generator. In *C++ Conference*, pages 87–102, 1990.

[90] Douglas C. Schmidt and Stephen D. Huston. C++ network programming: Mastering complexity using ace and patterns. *Addison-Wesley Longman*, 2002.

[91] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), 1998.

[92] Douglas C. Schmidt and Steve Vinoski. *The CORBA Component Model: Part 1, Evolving Towards Component Middleware.* Dr. Dobb's Journal. http://www.ddj.com/dept/cpp/184403884.

[93] B. Selic. An architectural pattern for real-time control software. *Selic, B. An Architectural Pattern for Real-time Control Software. In Vlissides, J.M., J.O. Coplien, and N.L. Kerth. (eds.) Pattern Languages of Program Design 2. Addison-Wesley, USA, 1996.*, 1996.

[94] Inc Silicon Graphics. *SGI's STL Programmer's Guide.* http://www.sgi.com/tech/stl/.

[95] Malcolm Spence. *CORBA Still Delivering !* http://www.omg.org/docs/omg/06-04-01.pdf.

[96] Richard W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, 1994. StEV w 94:1 1.Ex.

[97] David B. Stewart and Pradeep Khosla. Real-time scheduling of dynamically reconfigurable systems. In *IEEE International Conference on Systems Engineering*, pages 139–142, August 1991.

[98] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Professional; 3 edition, 2000.

[99] Clemens Szyperski. Component software: Beyond object-oriented programming (2nd. edition). *Addison-Wesley/ACM Press*, 2002.

[100] TimeSys. *TimeSys Linux 3.1.* http://www.timesys.com.

[101] Trolltech. *Qt3.* http://www.trolltech.com/.

[102] Lingfeng Wang. *Modern Industrial Automation Software Design.* John Wiley & Sons Inc (3 Mar 2006), 2006.

[103] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz. Total quality of service provisioning in middleware and applications. *The Journal of Microprocessors and Microsystems*, 27(2):45–54, March 2003.

[104] Shengquan Wang, Sangig Rho, Zhibin Mai, Riccardo Bettati, and Wei Zhao. Real-time component-based systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 428–437, 2005.

[105] L. Wills, S. Kannan, B. Heck, G. Vachtsevanos, C. Restrepo, S. Sander, D. Schrage, and J. Prasad. An open software infrastructure for reconfigurable control systems. *Proc. 19th Amer. Control Conf. (ACC-2000), Chicago, IL*, pages 2799–2803, 2000.

[106] S. Yau and C. Taweponsomkiat. Component configuration for object-oriented distributed real-time software development. *3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing - ISORC*, 2000.

[107] S. Yau and C. Taweponsomkiat. An approach to object-oriented component configuration for real-time software development. *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing - ISORC*, 2002.

[108] J. Zalewski. Real-time software architectures and design patterns: Fundamental concepts and their consequences. *Real-Time Programming. Pergamon, Oxford*, pages pp, 1–13, 1999.

[109] J. Zalewski. Real-time software design patterns. *9th Polish Conf. on Real-Time Systems Ustron, Poland*, pages pp, 16–19, September 2002.

[110] Lichen Zhang and Ruicheng Liu. Aspect-oriented real-time system modeling method based on uml. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005), 17-19 August 2005, Hong Kong, China*, pages 373–376. IEEE Computer Society, 2005.

Apêndice A

Descritor XML de implantação

Código A.1: Exemplo do arquivo descritor de implantação

```
<Deployment:deploymentPlan
xmlns:Deployment="http://www.omg.org/Deployment"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/Deployment Deployment.xsd">
    <label>Report-DeploymentPlan</label>
    <UUID>Report_Plan_UUID_0001</UUID>
    <realizes>
        <label>Report-cid</label>
        <UUID>c0965470-7b83-11d9-9669-0800200c9a66</UUID>
        <specificType></specificType>
        <supportedType>IDL:FinancialReport:1.0</supportedType>
        <port>
            <name>sorting_receptacle</name>
            <specificType>IDL:Sorting:1.0</specificType>
            <supportedType>IDL:Sorting:1.0</supportedType>
            <provider>false</provider>
            <exclusiveProvider>false</exclusiveProvider>
            <exclusiveUser>true</exclusiveUser>
            <optional>false</optional>
            <kind>SimplexReceptacle</kind>
        </port>
        ...
    </realizes>

    <implementation id="ReportManager-mdd">
        <name>ReportManager-mdd</name>
        <source></source>
        <artifact>ReportManager_exec</artifact>
        <artifact>ReportManager_svnt</artifact>
    </implementation>
    ...

    <instance id="ReportManager-idd">
        <name>ReportManager-idd</name>
        <node>MainNode</node>
        <source></source>
        <implementation>ReportManager-mdd</implementation>
        <configProperty>
            <name>ComponentIOR</name>
            <value>
                <type>
                    <kind>tk_string</kind>
                </type>
                <value>
                    <string>ReportManager.ior</string>
                </value>
            </value>
        </configProperty>
        <configProperty>
            <name>RegisterNaming</name>
            <value>
                <type>
```

```
54                  <kind>tk_string</kind>
55              </type>
56              <value>
57                  <string>Report/ReportManager</string>
58              </value>
59          </value>
60      </configProperty>
61      <configProperty>
62          <name>orientation</name>
63          <value>
64              <type>
65                  <kind>tk_string</kind>
66              </type>
67              <value >
68                  <string>portrait</string>
69              </value>
70          </value>
71      </configProperty>
72  </instance>
73  ...
74
75  <connection>
76      <name>financialreport_sortingalgorithm_connection</name>
77      <internalEndpoint>
78          <portName>sorting_facet</portName>
79          <kind>Facet</kind>
80          <instance>SortingAlgorithm-idd</instance>
81      </internalEndpoint>
82      <internalEndpoint>
83          <portName>sorting_receptacle</portName>
84          <kind>SimplexReceptacle</kind>
85          <instance>FinancialReport-idd</instance>
86      </internalEndpoint>
87  </connection>
88  ...
89
90  <artifact id="ReportManager_exec">
91      <name>ReportManager_exec</name>
92      <source></source>
93      <node></node>
94      <location>ReportManager_DnC_exec</location>
95      <execParameter>
96          <name>entryPoint</name>
97          <value>
98              <type>
99                  <kind>tk_string</kind>
100             </type>
101             <value>
102                 <string>create_ReportManagerHome_Impl</string>
103             </value>
104         </value>
105     </execParameter>
106 </artifact>
107
108 <artifact id="ReportManager_svnt">
109     <name>ReportManager_svnt</name>
110     <source></source>
111     <node></node>
112     <location>ReportManager_DnC_svnt</location>
113     <execParameter>
114         <name>entryPoint</name>
115         <value>
116             <type>
117                 <kind>tk_string</kind>
118             </type>
119             <value>
120                 <string>create_ReportManagerHome_Servant</string>
121             </value>
122         </value>
123         </execParameter>
124 </artifact>
125 ...
126
```

127 </**Deployment**:**deploymentPlan**>

O bloco *realizes*, nas linhas 8 a 24, contém as principais informações a respeito do Plano de Implantação, tais como o nome do plano (label), um identificador (*Universal Unique IDentifier* - UUID[1]) e a indicação dos *ports* (linhas 13 a 22) a serem utilizados naquela montagem. O nome do plano pode ser escolhido pelo desenvolvedor, porém é comum a presença do sufixo cid (*Component Interface Description*).

O bloco *implementation*, nas linhas 26 a 31, define um novo tipo de componente. Esse tipo será utilizado posteriormente, no bloco *instance*, para a criação de instâncias de componentes desse tipo. Cada novo tipo declarado possui um identificador (*id*) escolhido pelo desenvolvedor, porém é comum a presença do sufixo mdd (*Monolithic Deployment Description*). Nesse bloco são informados um nome para o tipo e os nomes dos blocos de artefatos (a serem descritos adiante). Para cada tipo de componente utilizado na aplicação deve existir um bloco *implementation* correspondente.

O bloco *instance*, nas linhas 34 a 72, indica a criação de uma nova instância de componente. Cada nova instância declarada possui um identificador (*id*) escolhido pelo desenvolvedor, porém é comum a presença do sufixo idd (*Instance Deployment Description*). Nesse bloco são informados um nome para a instância, o nó no qual esta instância será criada (para implantações remotas), o tipo da instância[2] e os valores iniciais dos atributos dessa instância. Dentre esses atributos, dois deles são implicitamente definidos em cada componente: *ComponentIOR* e *RegisterNaming*. O atributo *ComponentIOR* contém o arquivo no qual será gravada a referência remota (*Interoperable Object Reference* - *IOR*) para esse componente. O atributo *RegisterNaming* contém o caminho para registro desse componente no servidor de nomes. Esses dois atributos representam alternativas para que clientes possam descobrir e se comunicar com essa instância de componente. Atributos definidos no arquivo IDL podem ser inicializados através do bloco *configProperty* (linhas 61 a 71). A *tag name* desse bloco deve conter um o nome de um dos atributos definidos para esse componente no arquivo IDL. As *tags type* e *value* indicam o tipo e o valor do atributo, respectivamente. Para cada instância a ser criada na aplicação deve existir um bloco *instance* correspondente.

O bloco *connection*, nas linhas 75 a 87, indica a conexão de uma faceta com um receptáculo ou de um produtor com um depósito de eventos. Cada conexão é definida por um nome (*tag name*) e por dois participantes, denominados *internalEndpoints*. Para cada *internalPoint* indica-se o nome do *port* (deve ser o nome de um dos *ports* definidos para o componente no arquivo IDL), o tipo (faceta, receptáculo, produtor ou depósito) e a instância que contém aquele *port* e que participará da conexão[3]. Para cada conexão presente na aplicação deve existir um bloco *connection* correspondente.

Os blocos *artifact*, nas linhas 90 a 106 e 108 a 124, informam detalhes técnicos necessários para a

[1] Esses UUID's podem ser gerados através de utilitários tais como o *uuidgen*.

[2] O tipo da instância é indicado através da *tag* <*implementation*> e deve ser igual ao *id* de um tipo definido em algum bloco *implementation*.

[3] O nome da instância participante é indicado através da *tag* <*instance*> e deve ser igual ao *id* de uma instância definida em algum bloco *instance*.

implantação e execução dos componentes. Cada artefato é identificado por um *id*, o qual deve coincidir com o valor da *tag artifact* no bloco *implementation*. Dentre as informações desse bloco, destacam-se a *tag location*, utilizada para indicar a biblioteca que contém a implementação do componente, e a *tag entryPoint*, utilizada para indicar o método utilizado para criar os *homes* desses componentes. Para cada tipo de componente utilizado na aplicação deve existir um par de blocos *artifact*, um para indicar informações do executor criado pelo desenvolvedor e outro para indicar informações do *servant* gerado pelas ferramentas de compilação de arquivos CIDL. Para cada tipo de componente presente na aplicação deve existir um par de blocos *artifact* correspondentes.

Apêndice B

MPC (Make Project Creator)

O código B.1 apresenta um exemplo de um arquivo de configuração do MPC. Nesse arquivo, são definidos projetos e, para cada projeto, os arquivos IDL, CIDL e de código-fonte participantes.

Código B.1: Exemplo do arquivo de configuração do MPC

```
1  project(Hello_DnC_stub): ciao_client_dnc {
2      sharedname = Hello_DnC_stub
3      idlflags += -Sc
4
5      IDL_Files { Hello.idl }
6      Source_Files { HelloC.cpp }
7      Header_Files { HelloC.h }
8      Inline_Files { HelloC.inl }
9  }
10
11 project(Hello_DnC_svnt) : ciao_servant_dnc {
12     after += Hello_DnC_stub
13     sharedname  = Hello_DnC_svnt
14     libs     += Hello_DnC_stub
15
16     idlflags  +=  -Sc
17
18     CIDL_Files { Hello.cidl }
19     IDL_Files { HelloE.idl }
20     Source_Files { HelloEC.cpp HelloS.cpp Hello_svnt.cpp }
21     Header_Files { HelloEC.h HelloS.h Hello_svnt.h }
22     Inline_Files { HelloEC.inl HelloS.inl }
23 }
24
25 project(Hello_DnC_exec) : ciao_component_dnc {
26     after    += Hello_DnC_svnt
27     sharedname = Hello_DnC_exec
28     libs       += Hello_DnC_stub \
29                   Hello_DnC_svnt \
30
31     idlflags  +=  -Sc
32
33     IDL_Files { }
34     Source_Files { Hello_exec.cpp }
35     Header_Files { Hello_exec.h }
36     Inline_Files { }
37 }
```

Esse arquivo de configuração MPC define três projetos: *Hello_DnC_stub* (linha 1), *Hello_DnC_svnt* (linha 11) e *Hello_DnC_exec* (linha 26). Para cada projeto será gerado um arquivo *makefile* e uma biblioteca compartilhada (arquivo .so) correspondentes. Esses projetos definem bibliotecas para o *stub*, *skeleton* e executor do componente *Hello*, respectivamente. O MPC define projetos previamente configu-

rados, dos quais novos projetos podem "herdar" configurações básicas tais como: bibliotecas fundamentais do CIAO a serem utilizadas, parâmetros *default* para os compiladores de arquivos IDL e CIDL, dentre outras. Desta forma, o projeto *Hello_DnC_stub* "herda" as configurações do projeto *ciao_client_dnc*, que define todas as informações necessárias para a construção de *stubs*. De forma semelhante, os projetos *Hello_DnC_stub* e *Hello_DnC_exec* são também configurados a partir de projetos disponibilizados pelo MPC. Os projetos previamente configurados pelo MPC são arquivos com extensão .mpb e podem ser encontrados no diretório *${ACE_ROOT}/bin/MakeProjectCreator/config/*, onde *${ACE_ROOT}* é o diretório raiz da instalação do ACE.

Em cada projeto, palavras reservadas são utilizadas para indicar os arquivos a serem compilados, parâmetros de compilação e dependências entre projetos. A palavra reservada *sharedname* indica o nome da biblioteca compartilhada a ser gerada. Por exemplo, a linha 2 indica que a biblioteca compartilhada do projeto *Hello_DnC_stub* se chamará *libHello_DnC_stub.so*. A palavra reservada *idlflags* (linha 3) indica quais parâmetros serão utilizados no compilador de arquivos IDL. A palavra reservada *after* indica dependência entre execuções do *make*. Por exemplo, a linha 12 indica que o projeto *Hello_DnC_svnt* depende do projeto *Hello_DnC_stub*, ou seja, antes de compilar os arquivos do projeto *Hello_DnC_svnt*, o *makefile* gerado pelo MPC deverá compilar todos os arquivos do projeto *Hello_DnC_stub*, caso estes não estejam compilados. A palavra reservada *libs* indica quais bibliotecas devem ser *link-editadas* com a biblioteca do projeto em questão[1]. As palavras reservadas *IDL_Files*, *CIDL_Files*, *Source_Files*, *Header_Files* e *Inline_Files* indicam, respectivamente, os arquivos IDL, CIDL, fonte, *header* e de definições *inline* pertencentes a cada projeto.

Uma vez criado o arquivo de configuração do MPC, os *makefiles* podem ser gerados através da execução da ferramenta *mwc*, conforme indicado no código B.2. A ferramenta *mwc* recebe como parâmetro um arquivo de configuração MPC e gera um arquivo *makefile* para cada projeto definido.

Código B.2: Gerando os *makefiles* a partir de um arquivo de configuração do MPC

```
1 > mwc.pl <arquivo .mpc>
```

O MPC disponibiliza ainda um utilitário para geração automática de versões iniciais de arquivos de configuração. Conforme apresentado no código B.3, o utilitário *generate_component_mpc* gera um arquivo de configuração inicial contendo os projetos do *stub*, *skeleton* e *executor* do componente passado como parâmetro. Após a geração, o desenvolvedor pode adaptar esse arquivo às suas reais necessidades. A ferramenta *mwc* está localizada no diretório *${ACE_ROOT}/bin/*, enquanto o utilitário *generate_component_mpc* está localizado no diretório *${CIAO_ROOT}/bin/*, onde *${CIAO_ROOT}* é o diretório raiz da instalação do CIAO.

[1] Bibliotecas com símbolos indefinidos gerados pela ausência de *link-edição* com outras bibliotecas podem ser facilmente identificadas através do comando "ldd -r <arquivo .so>". As bibliotecas que contêm os símbolos indefinidos devem ser incluídas na palavra reservada *libs* do projeto.

Código B.3: Geração automática de um arquivo de configuração do MPC

```
1 > generate_component_mpc.pl <nome do componente>
```

Ressalta-se ainda que a ferramenta *mwc*, quando executada sem parâmetros, gera *makefiles* para todos os arquivos .mpc encontrados recursivamente a partir do diretório no qual a ferramenta foi executada. Essa funcionalidade é importante para a geração de um arquivo *makefile* que possibilita a compilação de um projeto grande, formado por vários sub-diretórios.

Apêndice C

Implementando um componente no CIAO

Este apêndice apresenta os passos necessários para a criação e implantação de um componente CIAO e para a implementação de um cliente para acesso ao serviço disponibilizado pelo componente. Essa atividade requer a completa instalação e compilação do ACE, TAO e CIAO, conforme apresentados no apêndice D. A implementação aqui descrita envolve a criação de um componente *Hello* simples, sem *ports* e com um atributo definido (*message*). Esse componente suporta uma interface com o método *hello()*, que imprime na tela o texto parametrizado pelo atributo do componente.

Implementação e compilação do componente

1. Geração do arquivo de configuração do MPC: escolha um diretório para a criação do componente e execute o comando '*generate_component_mpc.pl* <Nome do Componente>', por exemplo '*generate_component_mpc.pl Hello*'. Após esses comandos serão criados os arquivos *Hello.mpc*, *Hello_stub_export.h*, *Hello_svnt_export.h* e *Hello_exec_export.h*. O arquivo *Hello.mpc* já contém a definição de três projetos: o do *stub*, do *servant* e do executor (implementação do componente).

2. Criação do arquivo IDL do componente: crie um arquivo chamado <Nome do Componente>.idl (neste exemplo, Hello.idl) contendo as definições da interface suportada, do componente *Hello* e do componente home (*HelloHome*). Lembre-se que esse arquivo IDL deve incluir o arquivo *Components.idl*.

3. Criação do arquivo CIDL do componente: crie um arquivo chamado <Nome do Componente>.cidl (neste exemplo Hello.cidl) contendo as configurações do tipo de *container* a ser utilizado e do componente *home*. Lembre-se que esse arquivo CIDL deve incluir o arquivo *Hello.idl*.

4. Geração automática da versão inicial do executor: solicita-se uma compilação inicial para copiar a linha do compilador de arquivos cidl. Execute o comando '*mwc.pl*' para a geração dos arquivos

makefiles e, após, execute o comando '*make*'. Se tudo estiver correto, o arquivo IDL será compilado, os *stubs* serão compilados, a biblioteca dos *stubs* será gerada, o arquivo CIDL será compilado (linha que será copiada), os *servants* serão compilados e a a biblioteca dos *servants* será gerada. Na compilação do executor haverá um erro, pois este arquivo ainda não existe e será criado neste momento. Verifique a linha que contém o comando '*cidlc*' e copie-a inteiramente. Cole numa linha do terminal e acrescente o parâmetro '*–gen-exec-impl*' ao final da linha, porém antes de '– Hello.cidl'. Execute essa linha e serão gerados os arquivos Hello_exec.h e Hello_exec.cpp. Essa funcionalidade representa um grande avanço na produtividade e na qualidade de componentes implementados com o CIAO, pois poupa o desenvolvedor da codificação completa do executor do componente.

5. Implementação do componente: edite o arquivo *Hello_exec.cpp* e implemente os métodos que você definiu na interface do arquivo IDL. Limpe inteiramente o projeto (comando '*make realclean*') e compile novamente (comando '*make*'). A compilação deve ocorrer sem erros. As bibliotecas *libHello_stub.so*, *libHello_svnt.so* e *libHello_exec.so* deverão ter sido criadas e copiadas para o diretório *${ACE_ROOT}/lib*.

Implantação da montagem

1. Preparação para a implantação: crie um sub-diretório para realizar a implantação (geralmente denominado *descriptors*). Esse diretório conterá: o descritor de implantação, o mapa de nós do domínio de implantação e o *script perl* para implantação. Copie do diretório *${CIAO_ROOT}/examples/Hello/descriptors* os arquivos: *flattened_deploymentplan.cdp*, *NodeManagerMap.dat* e *run_test.pl*. O arquivo XML *flattened_deploymentplan.cdp* deve ser alterado para conter as informações sobre o seu componente. O arquivo *run_test.pl* deve ser alterado para a inclusão da linha '*readline STDIN*;' após a execução do *Plan Launcher*, de modo a aguardar que uma tecla seja pressionada antes de realizar a desimplantação. O arquivo *NodeManagerMap.dat* contém a lista dos nós de implantação.

2. Implantação: no diretório *descriptors*, execute o comando '*run_test.pl*'. A implantação deverá ser realizada com sucesso. Pressionando uma tecla qualquer, a montagem será desimplantada.

Implementação e execução do cliente

1. Implementação do cliente: copie o arquivo *starter.cpp* do diretório *${CIAO_ROOT}/examples/Hello/Sender/* para o diretório do seu componente e renomeie-o para *Cliente.cpp*.

2. Inclusão do projeto para compilação do cliente no arquivo de configuração do MPC: altere o arquivo *Hello.mpc* para incluir o projeto desse cliente e realize as devidas alterações no código.

3. Compilação do cliente: execute a geração dos arquivos *makefiles* através da re-execução do comando '*mwc.pl*'. Compile-o através da execução do comando '*make*'.

4. Execução do componentes: execute-o com o comando:
'*./Cliente -ORBInitRef NameService=corbaloc::localhost:60003/NameService*'.

Apêndice D

Instalando o CIAO e o ARCOS

Os passos requeridos para a compilação do CIAO são:

1. *Download* dos fontes do ACE, TAO e CIAO: o pacote contendo o ACE, TAO e CIAO pode ser obtido no *site http://deuce.doc.wustl.edu/Download.html*. A versão utilizada pelo ARCOS 0.5.1 é a *Latest Beta Kit* 0.5.1. O conteúdo desse pacote deve ser extraído em qualquer diretório do sistema de arquivos. No ambiente GNU/Linux sugere-se o uso do diretório */usr/local*, caso o usuário tenha permissões de *root*. O arquivo .tar.gz pode ser extraído através do WinZip, no ambiente Windows, ou através do comando 'tar -xzvf <arquivo.tar.gz>', no ambiente GNU/Linux.

2. Criação das variáveis de ambiente: a compilação e uso do CIAO requer a existência das variáveis de ambiente *ACE_ROOT*, *TAO_ROOT* e *CIAO_ROOT*. A variável *ACE_ROOT* deve conter o diretório no qual foi extraído o pacote dos fontes, incluindo o diretório raiz do ACE (*ACE_wrappers*). Por exemplo, no ambiente GNU/Linux, o conteúdo da variável *ACE_ROOT* seria */usr/local/ACE_wrappers/*. A variável *TAO_ROOT* deve conter o caminho para o diretório *TAO* presente no diretório especificado pela variável *ACE_ROOT*. Desta forma, a variável *TAO_ROOT* pode ser sempre ajustada como sendo ${*ACE_ROOT*}/*TAO* (ou %*ACE_ROOT*%/*TAO*, em ambientes Windows). A variável *CIAO_ROOT* deve conter o caminho para o diretório *CIAO* presente no diretório especificado pela variável *TAO_ROOT*. Desta forma, a variável *CIAO_ROOT* pode ser sempre ajustada como sendo ${*TAO_ROOT*}/*CIAO* (ou %*TAO_ROOT*%/*CIAO*, em ambientes Windows). No GNU/Linux, essas variáveis podem ser criadas através de inclusões de comandos '*export* <*variável*>=<*valor*>' no arquivo de configuração */etc/profile*. Inclui-se, portanto, ao final do arquivo */etc/profile*, os comandos '*export ACE_ROOT=/usr/local/ACE_wrappers*', '*export TAO_ROOT=${ACE_ROOT}/TAO*' e '*export CIAO_ROOT=${TAO_ROOT}/CIAO*'. Além disso, deve-se incluir os novos caminhos para as bibliotecas compartilhadas (arquivos .so) do ACE, TAO e CIAO através do inserção do comando '*export LD_LIBRARY_PATH =${LD_LIBRARY_PATH}:${ACE_ROOT}/lib:${ACE_ROOT}/ace*', também no arquivo */etc/-*

profile. Ainda nesse arquivo, a variável *PATH* deve ser alterada para conter os diretório dos executáveis do ACE, TAO e CIAO, através da inserção do comando '*export PATH=${PATH}:${ACE_ROOT}/bin:${CIAO_ROOT}/bin*'. Em ambientes Windows, as variáveis de ambientes são criadas/alteradas no Painel de Controle.

3. Configuração da plataforma alvo da compilação: o ACE, TAO, CIAO e ARCOS são soluções portáveis para uma série de plataformas. Desta forma, é necessário indicar a plataforma alvo na qual será realizada a compilação. Para isso, o ACE requer a criação de dois arquivos: *${ACE_ROOT}/ace/config.h* e *${ACE_ROOT}/include/makeinclude/platform_macros.GNU*. O arquivo *${ACE_ROOT}/ace/config.h* deve incluir um arquivo *header* contendo configurações da plataforma em questão. O diretório *${ACE_ROOT}/ace/* contém uma série de *headers* para as mais diversas plataformas. Para a compilação no GNU/Linux, o arquivo *${ACE_ROOT}/ace/config.h* deve conter a linha '*#include "ace/config-linux.h"*'. O arquivo *${ACE_ROOT}/include/makeinclude/platform_macros.GNU* deve também incluir um arquivo para a plataforma em questão. O diretório *${ACE_ROOT}/include/makeinclude/* contém uma série de arquivos de configuração para as mais diversas plataformas. Para a compilação no GNU/Linux, o arquivo *${ACE_ROOT}/include/makeinclude/platform_macros.GNU* deve conter as linhas '*no_hidden_visibility=1*' e '*include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU*'.

4. Geração dos arquivos *makefile*: uma vez configurada a plataforma alvo, os arquivos *makefiles* para a compilação do ACE, TAO e CIAO devem ser gerados. Para isso, o utilitário MPC (*Make Project Creator*), apresentado na sub-seção 3.3.4, é utilizado. No diretório *${ACE_ROOT}*, executa-se o comando 'mwc.pl ACE.mwc'. No diretório *${TAO_ROOT}*, executa-se o comando 'mwc.pl TAO.mwc'. No diretório *${CIAO_ROOT}*, executa-se o comando 'mwc.pl TAO_DAnCE.mwc'. Para a compilação no Windows, o comando 'mwc.pl' deve ser seguido do parâmetro '*-type vc8*', para que sejam gerados projetos para o Visual C++ 8.

5. Compilação do ACE e do TAO: após a geração dos *makefiles*, o ACE e o TAO podem ser compilados através da execução do comando '*make*' nos diretórios *${ACE_ROOT}* e *${TAO_ROOT}*, nesta ordem. No ambiente Windows, pode-se utilizar o Visual C++ ou uma versão Windows das ferramentas GNU para compilação em C++, denominada MinGW [64].

6. Instalação do compilador de arquivos CIDL: devido ao fato de requerer uma compilação mais trabalhosa, o compilador de arquivos CIDL do CIAO é disponibilizado, em versões binárias, pelo *DOC Group*. O compilador deve ser obtido do endereço *http://www.dre.vanderbilt.edu/cidlc/binary/*, observando a correspondência da versão do compilador com a versão do CIAO. O arquivo obtido deve ser descompactado (no ambiente GNU/Linux através do comando '*bunzip2 <pacote.bz2>*') e copiado para o diretório *${CIAO_ROOT}/bin*. O compilador deve ser renomeado (ou um ata-

lho/*link* pode ser criado) de *cidlc-<versão>* para *cidlc*. No ambiente GNU/Linux, deve-se conceder as permissões de execução através do comando '*chmod 755 cidlc*'.

7. Instalação das bibliotecas requeridas pelo compilador de CIDC: a versão binária do compilador de arquivos CIDL requer a instalação de duas bibliotecas: *boost-regex* (para utilização de expressões regulares) e *boost-filesystem* (para utilização portável do sistema de arquivos). Em ambientes GNU/Linux baseados no Debian [36], essas bibliotecas podem ser facilmente instaladas através do comando '*apt-get install libboost-regex1.33.1 libboost-filesystem1.33.1*'.

8. Compilação do CIAO: após os passos anteriores, o CIAO pode ser compilado através da execução do comando 'make' no diretório *${CIAO_ROOT}*. No ambiente Windows, pode-se utilizar o Visual C++ ou o MinGW.

Após a completa instalação do ACE, TAO e CIAO, o ARCOS pode ser compilado através da realização dos seguintes passos:

1. *Download* dos fontes do ARCOS: o pacote contendo o código-fonte do ARCOS pode ser obtido no endereço *http://arcos.sourceforge.net*. Vale ressaltar que a versão do ARCOS deve corresponder à versão do CIAO instalada anteriormente. O conteúdo desse pacote deve ser extraído em qualquer diretório do sistema de arquivos. No ambiente GNU/Linux sugere-se o uso do diretório */usr/local*, caso o usuário tenha permissões de *root*. O arquivo .tar.gz pode ser extraído através do WinZip, no ambiente Windows, ou através do comando 'tar -xzvf <arquivo.tar.gz>', no ambiente GNU/Linux.

2. Criação da variável de ambiente: a correta compilação do ARCOS requer a criação da variável de ambiente *ARCOS_ROOT*. Essa variável deverá conter o caminho para o diretório no qual o pacote do ARCOS foi descompactado (por exemplo, no GNU/Linux o conteúdo dessa variável seria */usr/local/ARCOS*.

3. Geração dos arquivos *makefile*: para a geração desses arquivos o programa 'mwc.pl' deve ser executado, sem argumentos, no diretório ${*ARCOS_ROOT*}.

4. Compilação do ARCOS: realizados os procedimentos acima descritos, o ARCOS pode ser facilmente compilado através da execução do comando '*make*' no diretório ${*ARCOS_ROOT*}.

Apêndice E

Arquivos IDL

Módulo de Aquisição de Dados

Código E.1: Componente *DAISServer*

```
#ifndef ARCOSDAISSERVER_IDL
#define ARCOSDAISSERVER_IDL

#pragma prefix "ufba.br"

#include <DAISServer.idl>
#include <ARCOSDAISProviderBase.idl>
#include <Components.idl>
#include <ARCOSDAISAccessPoints.idl>

module ARCOS
{
   module DataAcquisition
   {
      interface IExtendedDAISServer : ::DAIS::Server
      {
          void remove_dataaccess_session (in string session_name);
          ::ARCOS::DataAcquisition::IDAISProviderBaseFacet
              get_dais_provider();
      };

      component DAISServer supports IExtendedDAISServer
      {
          uses ::ARCOS::DataAcquisition::IDAISDANodeHomeAccessPoint
                  dais_dataaccess_node_home_ap;   // Connection for DAIS tree building
          uses ::ARCOS::DataAcquisition::IDAISProviderBaseFacet
                  dais_provider;                  // Connection for DAIS data acquisition
          provides IDAISServerAccessPoint
                  dais_server_ap;                 // Connection for DAISDASessions
      };

      home DAISServerHome manages DAISServer
      {
      };
   };
};

#endif /* ARCOSDAISSERVER_IDL */
```

Código E.2: Componente *DAISDASession*

```
#ifndef ARCOSDAISDASESSION_IDL
#define ARCOSDAISDASESSION_IDL

#pragma prefix "ufba.br"

#include <DAISDASession.idl>
#include <ARCOSDAISDANode.idl>
```

```
8  #include <ARCOSDAISDAGroup.idl>
9  #include <ARCOSDAISServer.idl>
10 #include <Components.idl>
11
12 module ARCOS
13 {
14    module DataAcquisition
15    {
16       interface IDAISDASessionFacet : ::DAIS::DataAccess::Session
17       {
18          void set_status (in ::DAIS::SessionStatus new_status);
19       };
20
21       component DAISDASession supports IDAISDASessionFacet
22       {
23          uses ::ARCOS::DataAcquisition::IDAISServerAccessPoint
24                  dais_server_ap;                // Connected to the DAISServer
25          uses ::ARCOS::DataAcquisition::IDAISDANodeHomeAccessPoint
26                  dais_dataaccess_node_home_ap;  // Connected to the NodeHome
27          uses ::ARCOS::DataAcquisition::IDAISDAGroupHomeAccessPoint
28                  dais_dataaccess_group_home_ap; // Connected to the GroupHome
29       };
30
31       home DAISDASessionHome manages DAISDASession
32       {
33       };
34    };
35 };
36
37 #endif /* ARCOSDAISDASESSION_IDL */
```

Código E.3: Componentes *DAISDANodeHome* e *DAISDANodeIterator*

```
1  #ifndef ARCOSDAISDANODE_IDL
2  #define ARCOSDAISDANODE_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
7  #include <DAISDANode.idl>
8  #include <ARCOSDAISAccessPoints.idl>
9
10 module ARCOS
11 {
12    module DataAcquisition
13    {
14       exception DuplicatedResourceID {string reason;};
15       exception ParentResourceIDNotFound {string reason;};
16
17       interface IDAISDANodeHome : ::DAIS::DataAccess::Node::IHome
18       {
19          void add_node (in ::DAFIdentifiers::ResourceID id,
20                         in ::DAFIdentifiers::ResourceID parent,
21                         in string label, in string descrip,
22                         in ::DAFIdentifiers::ResourceID type,
23                         in boolean is_leaf)
24                         raises (::DAIS::Node::IHome::UnknownResourceID);
25          boolean next_n (in unsigned long iterator_number, in unsigned long n,
26                         out ::DAIS::Node::Descriptions nodes);
27          void remove_dataaccess_node_iterator (in unsigned long number,
28                         in ::DAFIdentifiers::ResourceID parent);
29       };
30
31       component DAISDANodeHome supports IDAISDANodeHome
32       {
33          provides IDAISDANodeHomeAccessPoint
34                         dais_dataaccess_node_home_ap;  // Connected to the DAISDASession
35       };
36
37       home DAISDANodeHomeHome manages DAISDANodeHome
38       {
39       };
```

```
40
41        interface IDAISDANodeIterator : ::DAIS::Node::Iterator
42        {
43            attribute unsigned long number;
44            void set_parent(in ::DAFIdentifiers::ResourceID parent_id);
45        };
46
47        component DAISDANodeIterator supports IDAISDANodeIterator
48        {
49            uses IDAISDANodeHomeAccessPoint
50                        dais_dataaccess_node_home_ap; // Connected to the NodeHome
51        };
52
53        home DAISDANodeIteratorHome manages DAISDANodeIterator
54        {
55        };
56    };
57 };
58
59 #endif /* ARCOSDAISDANODE_IDL */
```

Código E.4: Componentes *DAISDAGroupHome DAISDAGroupManager DAISDAGroupClock* e *DAISDAGroupEntryIterator*

```
1  #ifndef ARCOSDAISDAGROUP_IDL
2  #define ARCOSDAISDAGROUP_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
7  #include <DAISGroup.idl>
8  #include <DAISGroupEntry.idl>
9  #include <ARCOSDAISAccessPoints.idl>
10
11 module ARCOS
12 {
13    module DataAcquisition
14    {
15        interface IDAISDAGroupHome : ::DAIS::DataAccess::Group::IHome
16        {
17            void set_session_name (in string session_name);
18            string get_session_name ();
19            void remove_dataaccess_group(in string group_name);
20        };
21
22        component DAISDAGroupHome supports IDAISDAGroupHome
23        {
24            provides IDAISDAGroupHomeAccessPoint
25                        dais_dataaccess_group_home_ap; // Connected to the DAISDASession
26        };
27
28        home DAISDAGroupHomeHome manages DAISDAGroupHome
29        {
30        };
31
32        interface IDAISDAGroupClock
33        {
34            void activate();
35            void deactivate();
36        };
37
38        eventtype EAcquisition
39        {
40        };
41
42        component DAISDAGroupClock supports IDAISDAGroupClock
43        {
44            provides IDAISDAGroupClockAccessPoint dais_dataaccess_group_clock_ap;
45            publishes EAcquisition event_acquisition;
46        };
47
48        home DAISDAGroupClockHome manages DAISDAGroupClock
```

```
49        {
50        };
51
52        interface IDAISDAGroupManager : ::DAIS::DataAccess::Group::Manager
53        {
54            boolean next_n (in unsigned long iterator_number,
55                            in unsigned long n,
56                            out ::DAIS::DataAccess::GroupEntry::DetailedDescriptions entries);
57            void remove_dataaccess_group_entry_iterator(in unsigned long number);
58            void activate();
59            void deactivate();
60        };
61
62        component DAISDAGroupManager supports IDAISDAGroupManager
63        {
64            uses IDAISDAGroupHomeAccessPoint
65                 dais_dataaccess_group_home_ap;     // Connected to the GroupHome
66            provides IDAISDAGroupManagerAccessPoint
67                 dais_dataaccess_group_manager_ap; // Connection for DAGroupEntryIterators
68        };
69
70        home DAISDAGroupManagerHome manages DAISDAGroupManager
71        {
72        };
73
74        interface IDAISDAGroupEntryIterator : ::DAIS::DataAccess::GroupEntry::Iterator
75        {
76            attribute unsigned long number;
77        };
78
79        component DAISDAGroupEntryIterator supports IDAISDAGroupEntryIterator
80        {
81            uses IDAISDAGroupManagerAccessPoint
82                      dais_dataaccess_group_manager_ap;  // Connected to the DAGroupManager
83        };
84
85        home DAISDAGroupEntryIteratorHome manages DAISDAGroupEntryIterator
86        {
87        };
88    };
89 };
90
91 #endif /* ARCOSDAISDAGROUP_IDL */
```

Código E.5: Interface *IDAISProviderBaseFacet*

```
1  #ifndef ARCOSDAISPROVIDERBASE_IDL
2  #define ARCOSDAISPROVIDERBASE_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
7  #include <ARCOSDAISDANode.idl>
8  #include <DAISDAIO.idl>
9  #include <DAISCommon.idl>
10
11 module ARCOS
12 {
13    module DataAcquisition
14    {
15        interface IDAISProviderBaseFacet
16        {
17            void build_dais_tree (in ::ARCOS::DataAcquisition::DAISDANodeHome
18                                     dais_dataaccess_node_home);
19            void get_values (in ::DAIS::ItemID node_ids,
20                             out ::DAIS::DataAccess::IO::ItemStates item_states);
21            ::DAIS::ItemErrors set_values (in ::DAIS::DataAccess::IO::ItemUpdates updates);
22        };
23    };
24 };
25
26 #endif /* ARCOSDAISPROVIDERBASE_IDL */
```

Código E.6: Componente *DAISEthernetPLCProvider*

```
#ifndef ARCOSDAISETHERNETPLCPROVIDER_IDL
#define ARCOSDAISETHERNETPLCPROVIDER_IDL

#pragma prefix "ufba.br"

#include <Components.idl>
#include <ARCOSDAISProviderBase.idl>

module ARCOS
{
   module DataAcquisition
   {
      interface IEthernetPLCProviderFacet : IDAISProviderBaseFacet
      {
      };

      component DAISEthernetPLCProvider
      {
         provides ::ARCOS::DataAcquisition::IEthernetPLCProviderFacet dais_provider;
         attribute string plc_ip_address;
      };

      home DAISEthernetPLCProviderHome manages DAISEthernetPLCProvider
      {
      };
   };
};

#endif /* ARCOSDAISETHERNETPLCPROVIDER_IDL */
```

Código E.7: Componente *DAISSimulatedCarProvider*

```
#ifndef ARCOSDAISSIMULATEDCARPROVIDER_IDL
#define ARCOSDAISSIMULATEDCARPROVIDER_IDL

#pragma prefix "ufba.br"

#include <Components.idl>
#include <ARCOSDAISProviderBase.idl>

module ARCOS
{
   module DataAcquisition
   {
      interface ISimulatedCarProviderFacet : IDAISProviderBaseFacet
      {
      };

      component DAISSimulatedCarProvider
      {
         provides ::ARCOS::DataAcquisition::ISimulatedCarProviderFacet dais_provider;
         attribute float engine_power;
         attribute float fric;
         attribute float mass;
         attribute float sample_rate;
      };

      home DAISSimulatedCarProviderHome manages DAISSimulatedCarProvider
      {
      };
   };
};

#endif /* ARCOSDAISSIMULATEDCARPROVIDER_IDL */
```

Código E.8: Interfaces *AccessPoint*

```
#ifndef ARCOSDAISACCESSPOINTS_IDL
#define ARCOSDAISACCESSPOINTS_IDL

#pragma prefix "ufba.br"

module ARCOS
{
    module DataAcquisition
    {
        interface IDAISDANodeHomeAccessPoint
        {
        };
        interface IDAISDAGroupHomeAccessPoint
        {
        };
        interface IDAISDAGroupClockAccessPoint
        {
        };
        interface IDAISDAGroupManagerAccessPoint
        {
        };
        interface IDAISServerAccessPoint
        {
        };
     };
};
#endif /* ARCOSDAISACCESSPOINTS_IDL */
```

Módulo de Controle

Código E.9: Tipo de evento *EControlData*

```
#ifndef ARCOSCONTROLDATAEVENT_IDL
#define ARCOSCONTROLDATAEVENT_IDL

#pragma prefix "ufba.br"

#include <Components.idl>

module ARCOS {
    module Control {
        eventtype EControlData
        {
            public string control_data;
        };
    };
};

#endif /* ARCOSCONTROLDATAEVENT_IDL */
```

Código E.10: Componente *ControlManager*

```
#ifndef ARCOSCONTROLMANAGER_IDL
#define ARCOSCONTROLMANAGER_IDL

#pragma prefix "ufba.br"

#include <ARCOSControllerBase.idl>
#include <ARCOSControlManagerDAISCallback.idl>
#include <DAFIdentifiers.idl>
#include <ARCOSControlDataEvent.idl>
#include <Components.idl>

module ARCOS
{
    module Control
    {
        interface IControlManager
```

```
17        {
18            void activate();
19            void deactivate();
20        };
21
22        component ControlManager
23        {
24            uses ::ARCOS::Control::IControllerBaseFacet
25                        controller;                                    // The controller
26            uses ::ARCOS::Control::IControlManagerDAISCallbackAccessPoint
27                        control_manager_dais_callback_ap;              // The connected callback
28            consumes ::ARCOS::Control::EControlData control_data;
29            attribute unsigned short                sampling_rate; // ms
30            attribute string                        dais_server_name;
31            attribute ::DAFIdentifiers::ResourceID  sensor_dais_leaf;
32            attribute ::DAFIdentifiers::ResourceID  actuator_dais_leaf;
33            attribute string                        RegisterNaming;
34            attribute float                         setpoint;
35        };
36
37        home ControlManagerHome manages ControlManager
38        {
39        };
40    };
41 };
42
43 #endif /* ARCOSCONTROLMANAGER_IDL */
```

Código E.11: Componente *ControlManagerDAISCallback*

```
1  #ifndef ARCOSCONTROLMANAGERDAISCALLBACK_IDL
2  #define ARCOSCONTROLMANAGERDAISCALLBACK_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <ARCOSControlDataEvent.idl>
7  #include <DAISDAIO.idl>
8  #include <Components.idl>
9
10 module ARCOS
11 {
12    module Control
13    {
14        interface IControlManagerDAISCallbackAccessPoint {
15        };
16
17        component ControlManagerDAISCallback supports ::DAIS::DataAccess::IO::Callback
18        {
19            provides IControlManagerDAISCallbackAccessPoint
20                control_manager_dais_callback_ap;  // Connection for the ControlManager
21            publishes ::ARCOS::Control::EControlData
22                control_data;                      // Connected to the ControlManager
23        };
24
25        home ControlManagerDAISCallbackHome manages ControlManagerDAISCallback
26        {
27        };
28    };
29 };
30
31 #endif /* ARCOSCONTROLMANAGERDAISCALLBACK_IDL */
```

Código E.12: Interface *IControllerBaseFacet*

```
1  #ifndef ARCOSCONTROLLERBASE_IDL
2  #define ARCOSCONTROLLERBASE_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
```

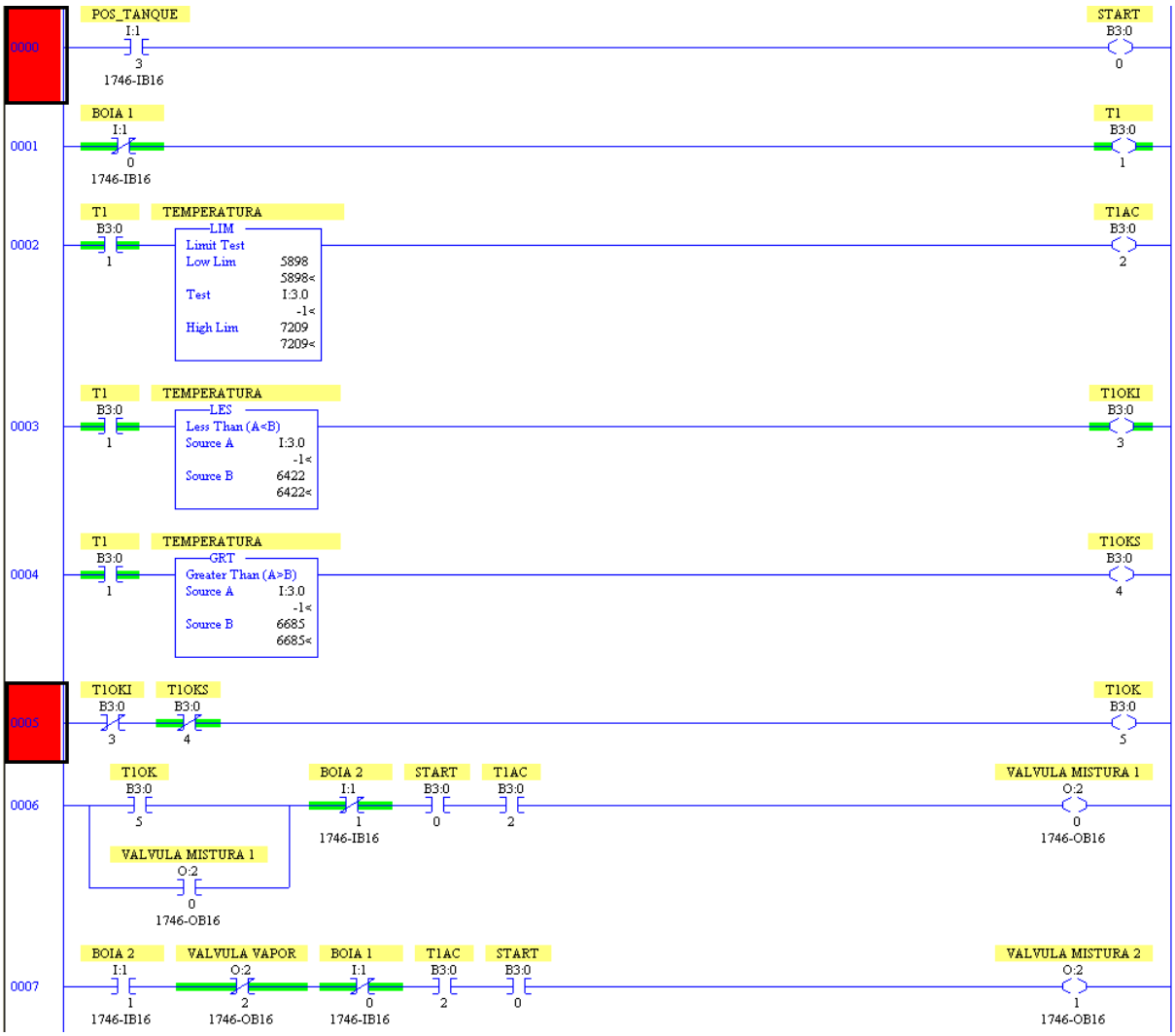
```
7
8  module ARCOS
9  {
10     module Control
11     {
12        interface IControllerBaseFacet
13        {
14           float control (in float error, in float control_old);
15        };
16     };
17 };
18
19 #endif /* ARCOSCONTROLLERBASE_IDL */
```

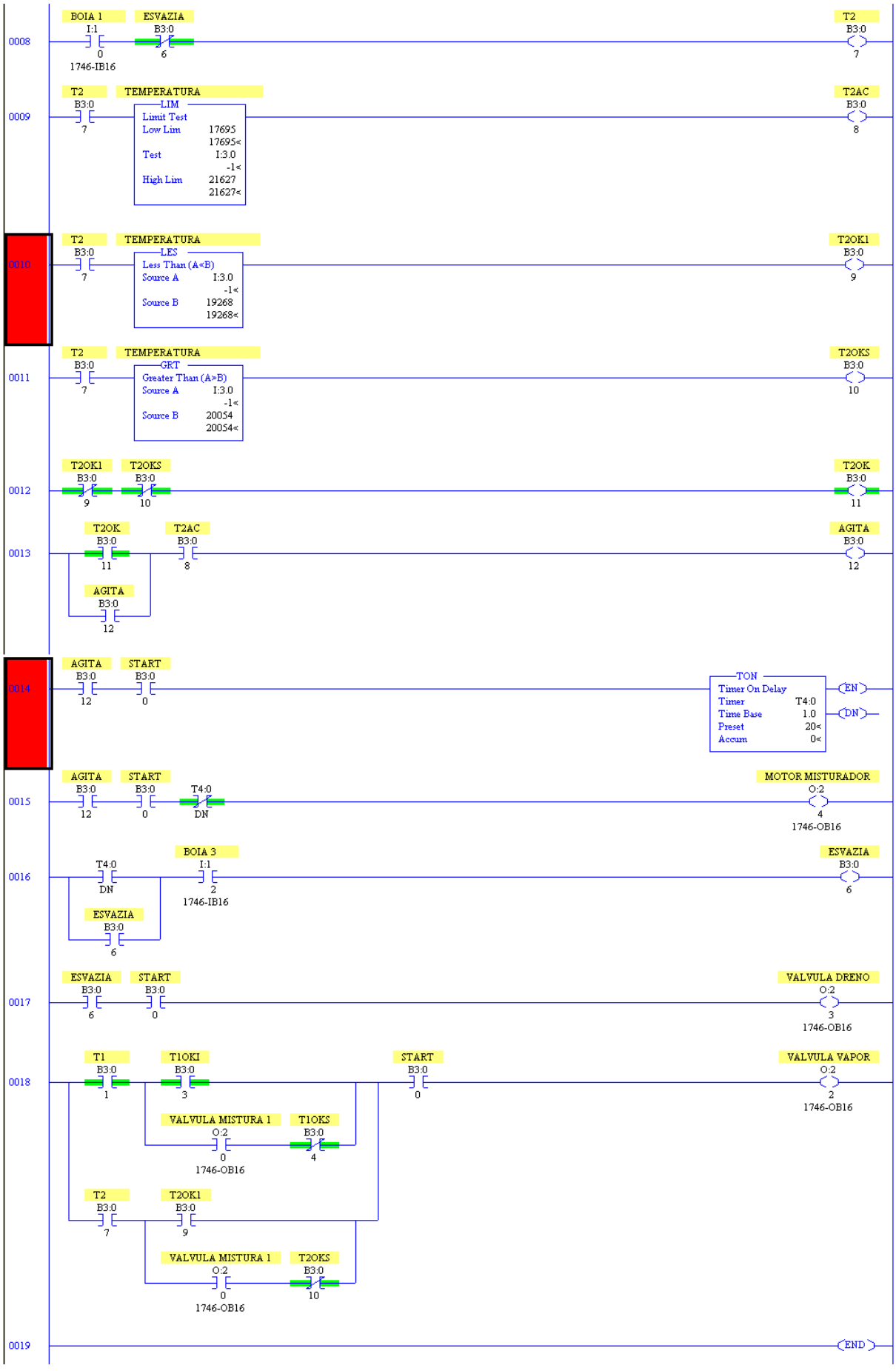
Código E.13: Componente *PIDController*

```
1  #ifndef ARCOSPIDCONTROLLER_IDL
2  #define ARCOSPIDCONTROLLER_IDL
3
4  #pragma prefix "ufba.br"
5
6  #include <Components.idl>
7  #include <ARCOSControllerBase.idl>
8
9  module ARCOS
10 {
11    module Control
12    {
13       interface IPIDControllerFacet : IControllerBaseFacet
14       {
15       };
16
17       component PIDController
18       {
19          provides ::ARCOS::Control::IPIDControllerFacet controller;
20          attribute float kp;
21          attribute float ki;
22          attribute float kd;
23          attribute unsigned short sampling_rate;
24       };
25
26       home PIDControllerHome manages PIDController
27       {
28       };
29    };
30 };
31
32 #endif /* ARCOSPIDCONTROLLER_IDL */
```

Apêndice F

Programa LADDER para supervisão do reator químico




0008
BOIA 1
I:1
0
1746-IB16
ESVAZIA
B3:0
6
T2
B3:0
7
0009
T2
B3:0
7
TEMPERATURA
LIM
Limit Test
Low Lim 17695
17695<
Test I:3.0
-1<
High Lim 21627
21627<
T2AC
B3:0
8
0010
T2
B3:0
7
TEMPERATURA
LES
Less Than (A<B)
Source A I:3.0
-1<
Source B 19268
19268<
T2OK1
B3:0
9
0011
T2
B3:0
7
TEMPERATURA
GRT
Greater Than (A>B)
Source A I:3.0
-1<
Source B 20054
20054<
T2OKS
B3:0
10
0012
T2OK1
B3:0
9
T2OKS
B3:0
10
T2OK
B3:0
11
0013
T2OK
B3:0
11
T2AC
B3:0
8
AGITA
B3:0
12
AGITA
B3:0
12
0014
AGITA
B3:0
12
START
B3:0
0
TON
Timer On Delay
Timer T4:0
Time Base 1.0
Preset 20<
Accum 0<
EN
DN
0015
AGITA
B3:0
12
START
B3:0
0
T4:0
DN
MOTOR MISTURADOR
O:2
4
1746-OB16
0016
T4:0
DN
BOIA 3
I:1
2
1746-IB16
ESVAZIA
B3:0
6
ESVAZIA
B3:0
6
0017
ESVAZIA
B3:0
6
START
B3:0
0
VALVULA DRENO
O:2
3
1746-OB16
0018
T1
B3:0
1
T1OKI
B3:0
3
VALVULA MISTURA 1
O:2
0
1746-OB16
T1OKS
B3:0
4
START
B3:0
0
VALVULA VAPOR
O:2
2
1746-OB16
T2
B3:0
7
T2OK1
B3:0
9
VALVULA MISTURA 1
O:2
0
1746-OB16
T2OKS
B3:0
10
0019
END

Apêndice G

Manual do ARCOS Management Tool

O *ARCOS Management Tool* (AMT) reúne um conjunto de aplicações voltadas para a utilização dos serviços disponibilizados pelo *Framework ARCOS* (componentes *server-side*), bem como para o auxílio ao desenvolvimento de novas aplicações baseadas no ARCOS. A figura 3.11 apresentou a tela inicial do AMT. Conforme destacado na figura G.1, o AMT está dividido em quatro módulos: o *DAIS Browser*, o *DAIS Manager*, o *ARCOS Applications* e o *Assembly Tool*.

Figura G.1: Módulos principais do ARCOS Management Tool

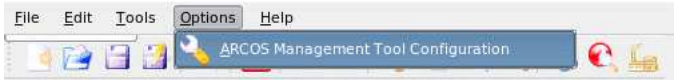
O *DAIS Browser* representa um cliente genérico para acesso a servidores DAIS, permitindo a visualização da árvore disponibilizada pelo servidor, a criação de grupos de aquisição com diferentes freqüências e a inclusão de itens nestes grupos. O *DAIS Manager* possibilita o gerenciamento de um servidor DAIS, visualizando as sessões atualmente criadas e os grupos criados em cada sessão. O *ARCOS Applications* apresenta as duas aplicações desenvolvidas neste trabalho: o sistema de supervisão do reator químico e o sistema para controle PID do piloto automático. O *Assembly Tool* auxilia o desenvolvimento de novas aplicações baseadas no ARCOS, à medida em que gera, automaticamente, versões iniciais de um novo *DAIS Provider* e de um novo controlador. Além disso, o descritor XML de implantação também é automaticamente gerado por essa ferramenta. Nas sessões seguintes serão apresentados como essas ferramentas podem ser utilizadas.

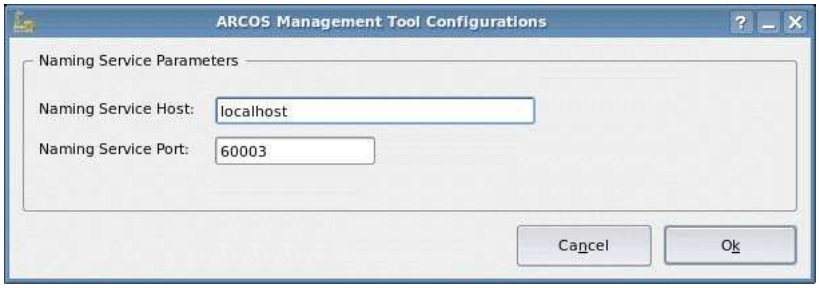
DAIS Browser

O DAIS Browser é um cliente genérico para acesso a servidores em conformidade com o padrão DAIS. As possíveis operações nessa aplicação são: conexão a um servidor DAIS e visualização da árvore, criação de um grupo de aquisição e inclusão de itens em grupos de aquisição.

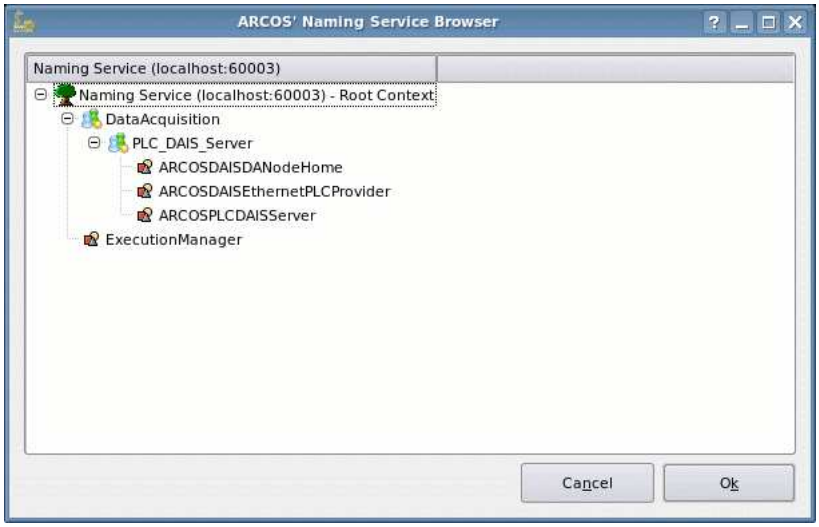
Os passos necessários para a conexão com um servidor DAIS são:

1. Verificar a correta localização do Servidor de Nomes através do menu "*Options*", item "*ARCOS Management Tool Configuration*". Na tela exibida, informar o *host* e a porta de execução do Servidor de Nomes.

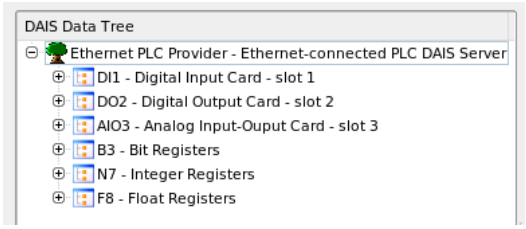




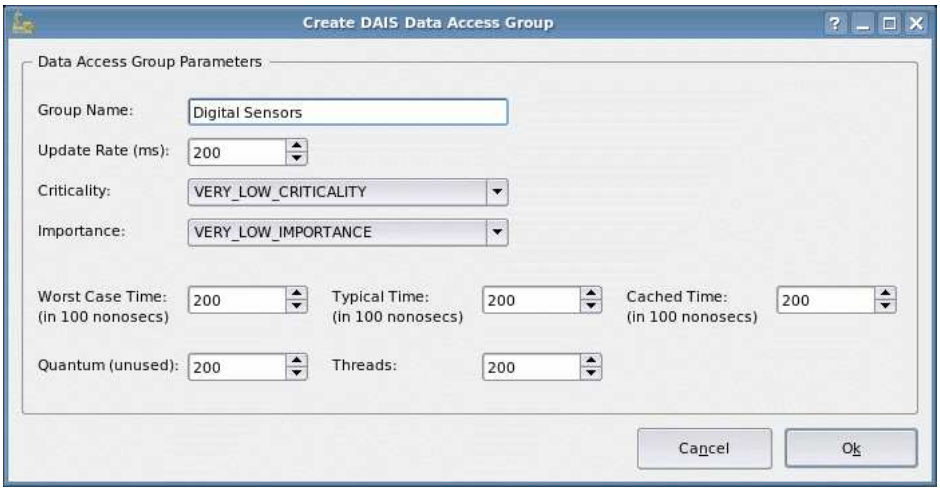
2. Escolher o servidor DAIS a ser conectado através da visualização do Servidor de Nomes. Ao pressionar o botão "*Connect*", o visualizador do Servidor de Nomes do AMT será exibido. O usuário deve localizar a referência para o servidor DAIS em questão e pressionar o botão "*Ok*".



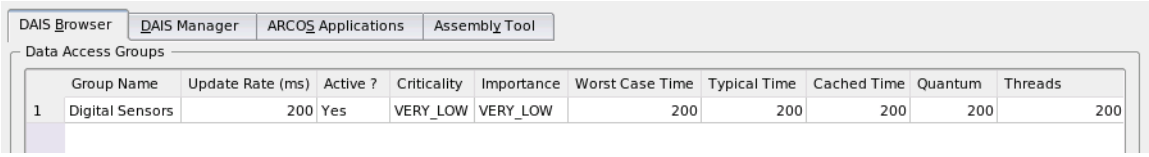
3. Após a escolha do servidor DAIS, a árvore disponibilizada pode ser visualizada no painel à esquerda do AMT.



4. Após o estabelecimento da conexão com o servidor DAIS, os grupos de aquisição de dados podem ser criados e folhas DAIS podem ser inseridas nestes grupos. Para a criação de um grupo de aquisição de dados, pressione o botão "*Create Group*" e informe os dados solicitados na tela de criação do grupo.

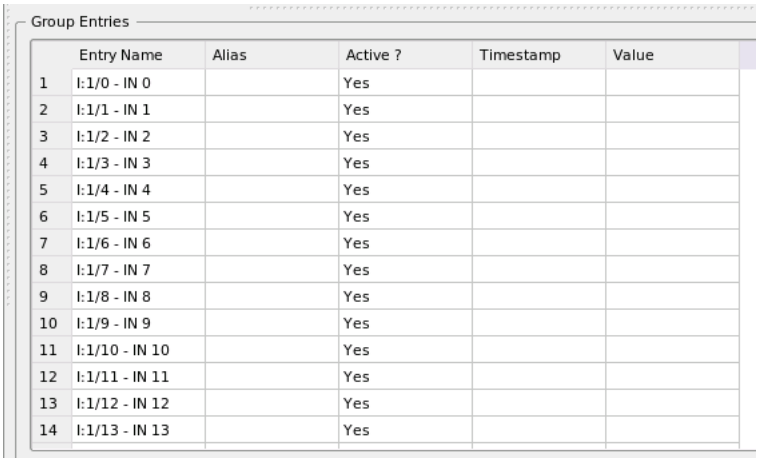


5. Cada grupo criado é exibido na tabela de grupos do *DAIS Browser*. Diversos grupos podem ser criados, cada um com as suas respectivas taxas de aquisição e os outros requisitos temporais.

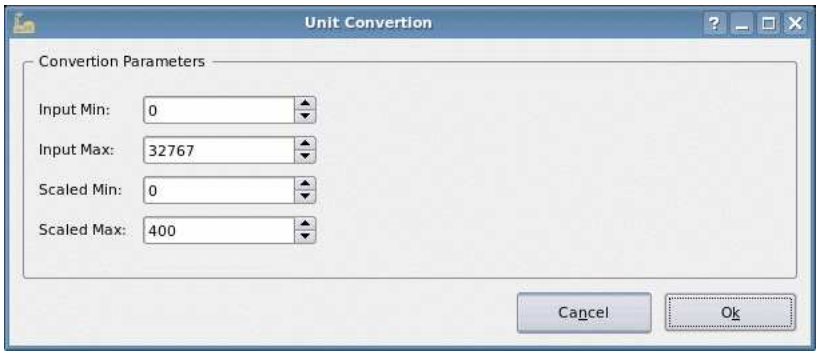


6. Os dados disponibilizados pelo servidor DAIS são representados pelas folhas exibidas na árvore de dados. O *DAIS Browser* permite a inclusão isolada de uma folha DAIS ou a inclusão recursiva de todas as folhas de um ramo da árvore. Para realizar a inclusão de folhas, selecione o grupo de aquisição no qual ocorrerá a inserção, seleciona a folha (ou ramo) desejado e pressione o botão "*Add Leaf as Entry*" (ou "*Add Branch as Entry*" caso um ramo seja selecionado). Após a inserção, as

folhas pertencentes a um determinado grupo são exibidas na tabela de entradas de grupo do *DAIS Browser*.



7. Para dados analógicos freqüentemente é necessário realizar conversões de escala, com o objetivo de transformar valores, da palavra analógica recebida, em valores que representam grandezas da planta. Por exemplo, na supervisão do reator químico a temperatura foi recebida como um valor entre 0 e 32767, quando na verdade desejamos representar a temperatura de 0^oC a 100^oC. Para isso, o *Server Browser* disponibiliza um conversor de unidades. Para utilizá-lo, o usuário deve selecionar o item do grupo a ser convertido e pressionar o botão "*Unit Convertion*".



8. Itens de grupo, bem como grupos de aquisição podem ser removidos através dos botões "*Remove Entry*" e "*Remove Group*", respectivamente. O botão "*Activate/Deactivate*" permite desativar temporariamente um grupo ou item de grupo, fazendo com que o servidor não transmita mais informações sobre esse grupo ou item de grupo.

DAIS Manager

O *DAIS Manager* é um aplicativo simples, com objetivos de monitoração do estado do servidor DAIS através da visualização das sessões de acesso a dados criadas e dos grupos de aquisição criados em cada uma destas sessões. Uma outra funcionalidade importante do *DAIS Manager* é a ativação do Canal de Eventos de Tempo-Real, iniciando o processo de transmissão dos dados coletados da planta. Uma vez que todos os clientes estejam devidamente conectados, com grupos de aquisição criados e com as folhas DAIS incluídas em grupos, a ativação do Canal de Eventos de Tempo-Real inicia a transmissão de dados para todos estes clientes. Essa restrição é dada pela execução *off-line* do Serviço de Eventos de Tempo-Real do TAO.

ARCOS Applications

O módulo "*ARCOS Applications*" do *ARCOS Management Tool* contém as duas aplicações desenvolvidas para a validação da plataforma proposta: o sistema de supervisão do reator químico e o sistema de controle PID para piloto automático.

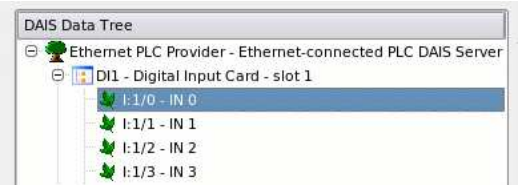
Supervisão do reator químico

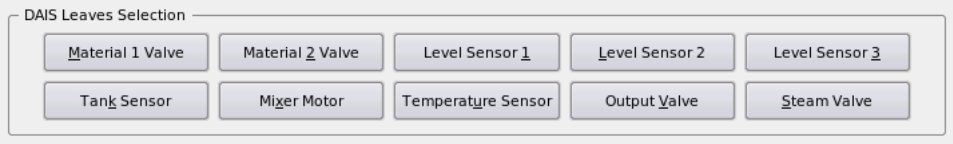
Os passos necessários para a utilização do supervisório específico para o reator químico são:

1. Realizar a implantação da montagem que utiliza o componente *DAISEthernetPLCProvider*. Para isso, o *script run_test_plc.pl*, localizado no diretório *${ARCOS_ROOT}/descriptors/* deve ser executado.

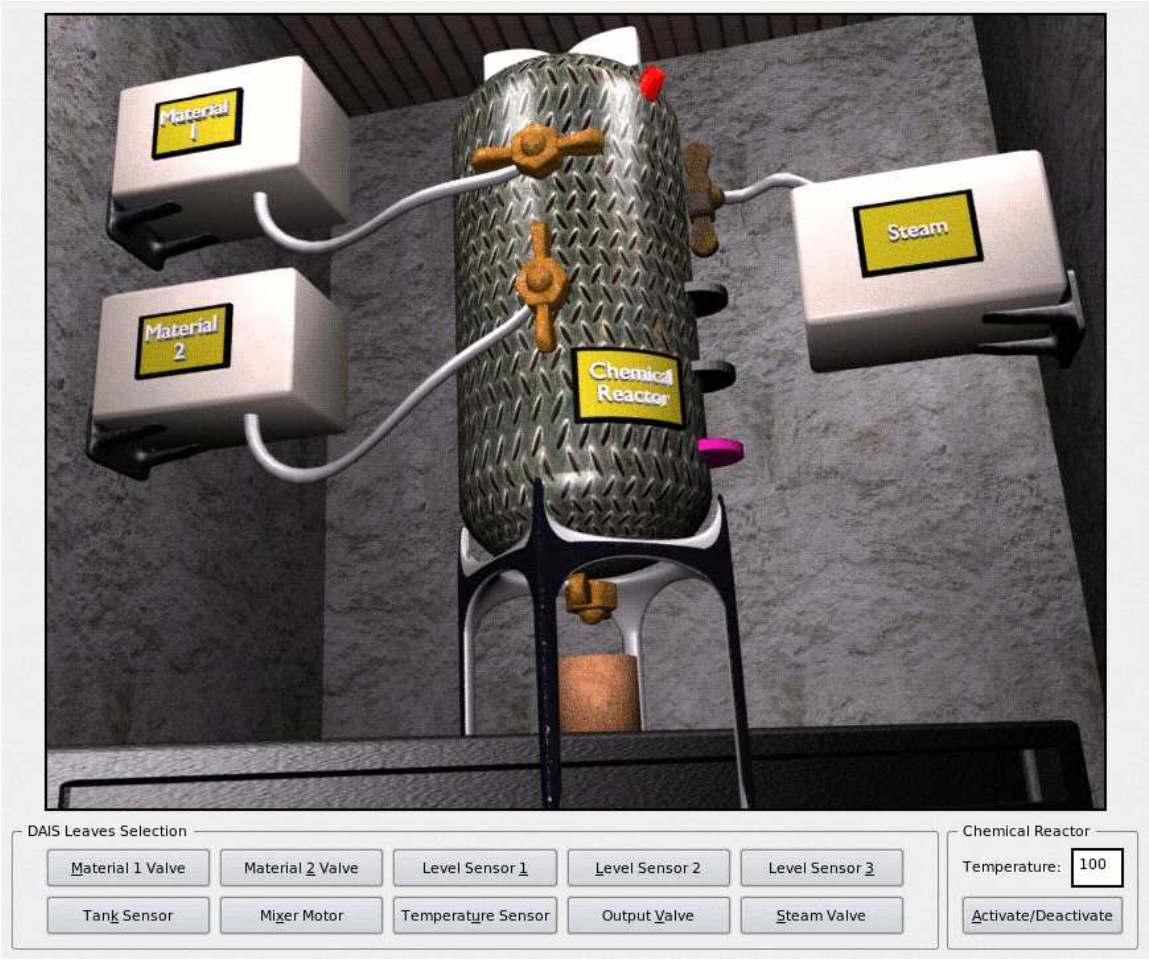
2. Realizar a conexão com o servidor DAIS. Pressionar o botão "*Connect*" e selecionar o servidor DAIS que representa o CLP *Allen-Bradley SLC 5/05* (caminho DataAcquisition → PLC_DAIS_Server → ARCOSPLCDAISServer).

3. Realizar as ligações das folhas DAIS com as representações gráficas dos sensores e atuadores. No módulo "ARCOS Applications", sub-módulo "*Chemical Reactor*", selecionar a folha DAIS que representa o sensor/atuador e pressionar o botão correspondente no grupo "*DAIS Leaves Selection*". Esse procedimento deve ser realizado para todos os dez sensores/atuadores utilizados, conforme ligações apresentadas na figura 4.9.





4. Ativar o supervisório. Após a realização de todas as ligações o supervisório deve ser ativado, pressionado o botão "*Activate/Deactivate*". A partir desse momento, as operações realizadas no *kit* didático são automaticamente refletidas em animações 3D realizadas pelo *ARCOS Management Tool*. A temperatura é visualizada no canto inferior direito do supervisório.



Controle PID para piloto automático

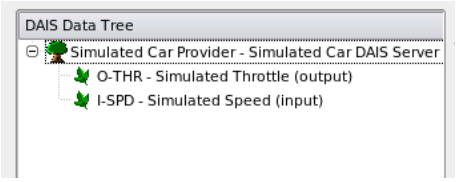
Os passos necessários para a utilização do sistema de controle PID para piloto automático são:

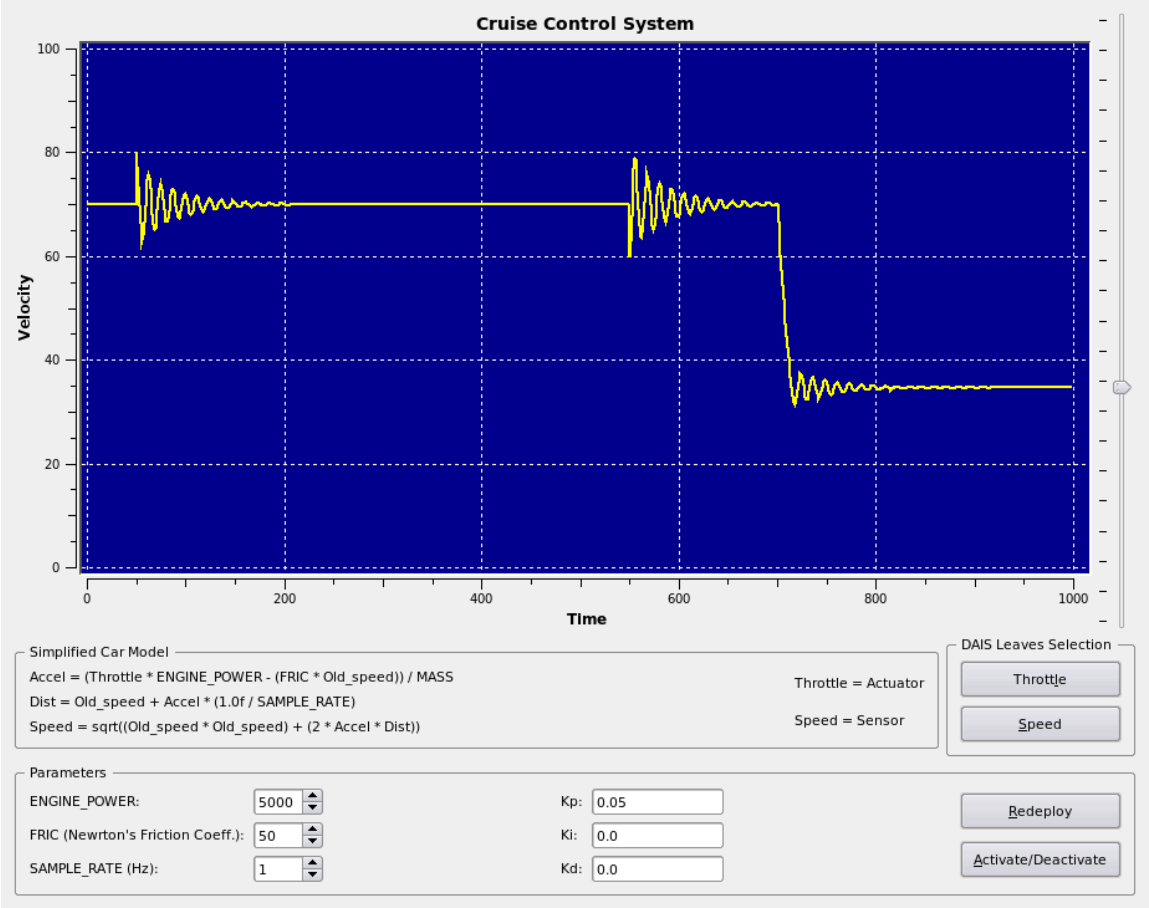
1. Realizar a implantação da montagem que utiliza os componentes *DAISSimulatedCarProvider* e *PIDController*. Para isso, o *script run_test_car+control.pl*, localizado no diretório *${ARCOS_ROOT}/*

descriptors/ deve ser executado.

2. Realizar a conexão com o servidor DAIS. Pressionar o botão "*Connect*" e selecionar o servidor DAIS que representa o veículo simulado (caminho DataAcquisition → Car_DAIS_Server → ARCOSCarDAISServer).

3. Realizar as ligações das folhas DAIS que representam o sensor de velocidade e o acelerador do veículo. No módulo "ARCOS Applications", sub-módulo "*PID Control*", selecionar a folha DAIS que representa o sensor de velocidade e pressionar o botão "*Speed*" no grupo "*DAIS Leaves Selection*". Neste momento, será solicitada a referência para o componente *ControlManager*. Navegue no visualizador do Servidor de Nomes e informe a referência solicitada. Selecionar a folha DAIS que representa o acelerador e pressionar o botão "*Throttle*" no grupo "*DAIS Leaves Selection*".





4. Ativar o supervisório. Após a realização das ligações o supervisório deve ser ativado, pressionando

o botão "*Activate/Deactivate*". Neste momento, será solicitada a referência para o componente *PIDController*. Navegue no visualizador do Servidor de Nomes e informe a referência solicitada. Após esta operação, o *setpoint* que representa a velocidade a ser mantida pode ser ajustada através do *slider* no canto direito da tela. O grupo de botões "*Parameters*" permite a alteração dos parâmetros do modelo do veículo e dos parâmetros k_{p}, k_{i} e k_{d} do controlador. Sempre que esses valores forem alterados o botão "*Redeploy*" deve ser pressionado para que os componentes sejam reconfigurados.

Assembly Tool

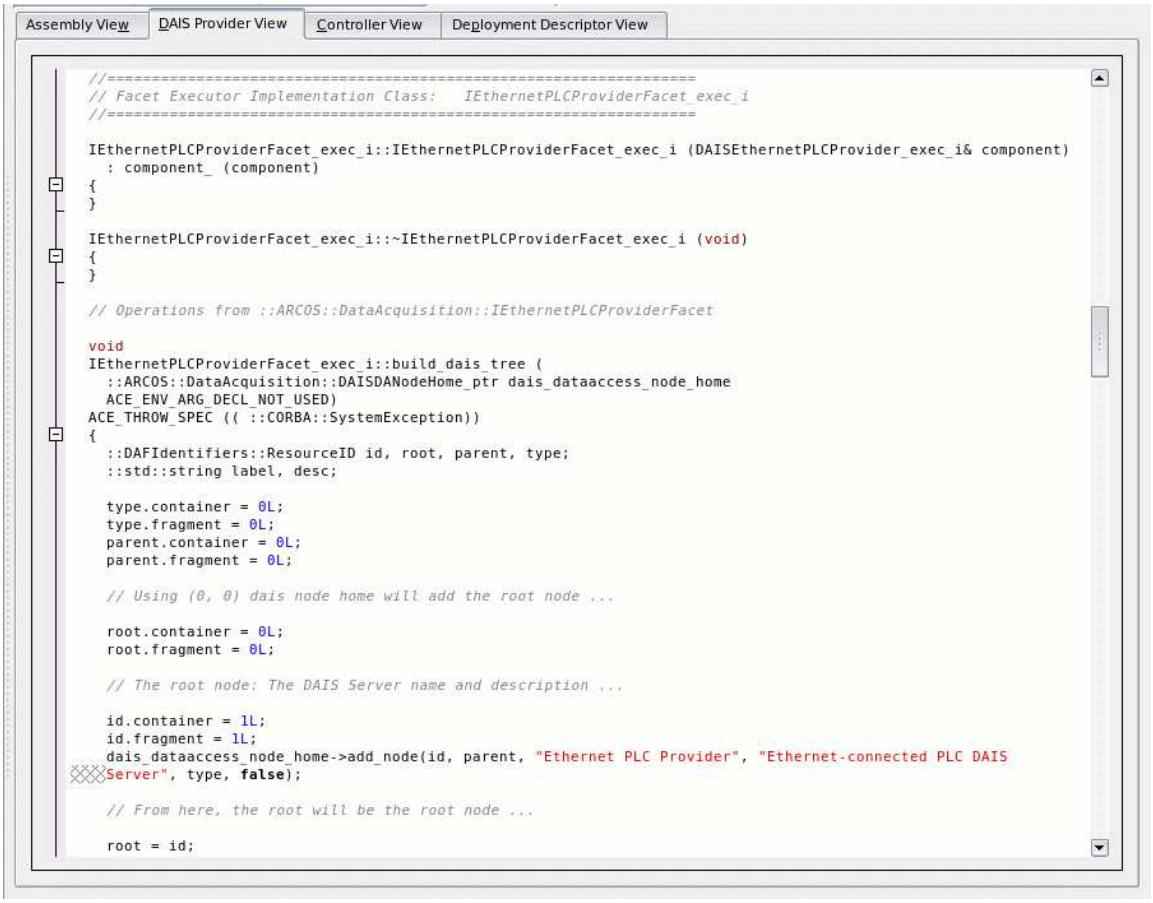
O *ARCOS Assembly Tool* (AST) é uma ferramenta de suporte ao desenvolvimento de novas aplicações industriais baseadas no ARCOS. Ela orienta o desenvolvedor na realização dos passos requeridos para especializar a plataforma para uma dada situação de aquisição de dados e controle. Dentre as suas funcionalidades, destaca-se a criação de versões iniciais de novos *DAIS Providers* e controladores, bem como a geração do descritor XML de implantação.

O AST disponibiliza quatro visões da nova montagem a ser criada: a visão gráfica da montagem (*Assembly View*), a visão do *DAIS Provider* (*DAIS Provider View*), a visão do controlador (*Controller View*) e a visão do descritor XML de implantação (*Deployment Descriptor View*).

Os passos a serem executados para a criação de um novo *DAIS Provider* e um novo controlador são:

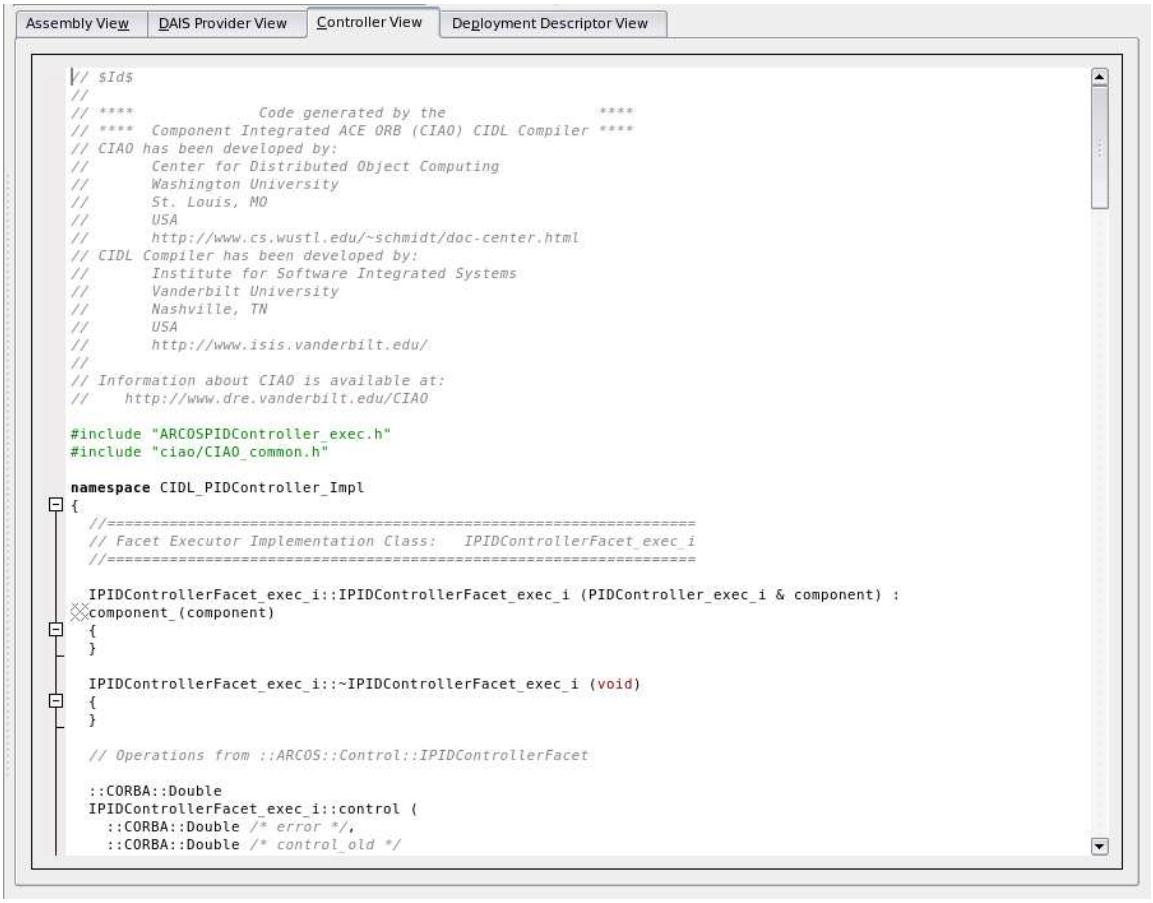
1. Criar o novo *DAIS Provider*: na tela principal do AST, pressionar o botão "*Create DAIS Provider*" e informar o nome do novo *DAIS Provider*.

2. Implementar o novo *DAIS Provider*: quando é solicitada a criação de um novo *DAIS Provider*, o AST gera automaticamente uma implementação inicial deste *provider*, requerendo que o desenvolvedor implemente somente os métodos *build_dais_tree()*, *get_value()* e *set_values()*. Esses métodos podem ser diretamente implementados na visão do *DAIS Provider*, disponibilizada pelo AST. O novo *provider* pode ser compilado pressionando o botão "*Compile*".

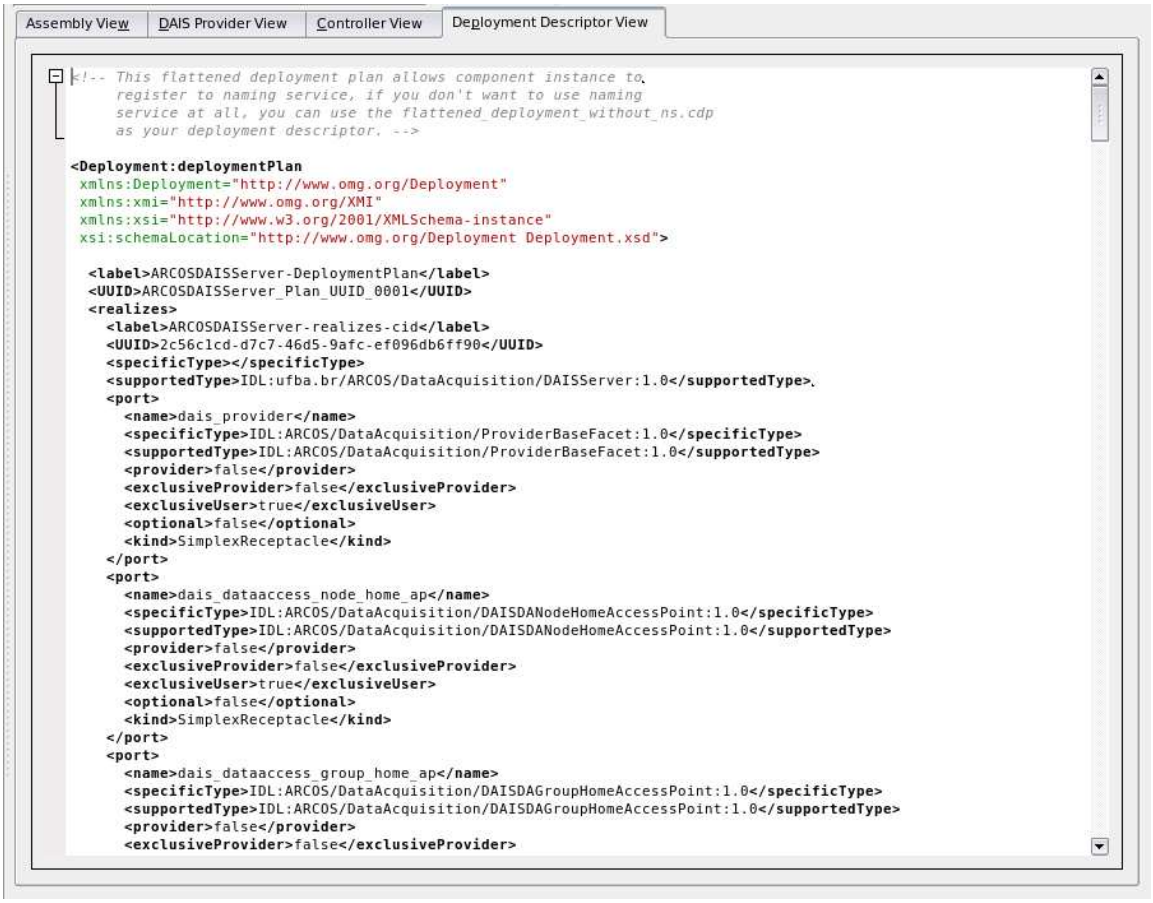


3. Criar o novo controlador: na tela principal do AST, pressionar o botão "*Create Controller*" e informar o nome do novo controlador.

4. Implementar o novo controlador: quando é solicitada a criação de um novo controlador, o AST gera automaticamente uma implementação inicial deste componente, requerendo que o desenvolvedor implemente somente o método *control()*. Esse método pode ser diretamente implementado na visão do controlador, disponibilizada pelo AST. O novo controlador pode ser compilado pressionando o botão "*Compile*".



5. Verificar o descritor de implantação: o AST gera automaticamente um descritor XML de implantação que conecta o novo *DAIS Provider* e controlador às estruturas internas do ARCOS. Esse descritor pode ser alterado na visão correspondente do AST, para comportar pequenas mudanças.



6. Implantar a nova montagem: a nova montagem pode ser implantada pressionando o botão "*Deploy*". Após esses passos o *DAIS Server Browser* pode ser utilizado para monitorar a aquisição do novo *DAIS Provider*.