3
Arquitetura do Sistema

Este capítulo irá descrever a arquitetura geral do sistema, justificando as decisões de implementação tomadas. Na primeira seção iremos considerar um conjunto de nós interagindo para formar a rede LuaPS. Após a definição da arquitetura de rede que será utilizada, iremos considerar um único nó da rede e discutir a arquitetura em camadas do sistema.

3.1
Rede LuaPS

Ao considerarmos sistemas publish-subscribe existentes, podemos perceber que existem duas formas básicas de definir a rede publish-subscribe. Ao longo da seção iremos explicar as duas arquiteturas, destacar vantagens e desvantagens de cada uma e definir a arquitetura escolhida.

A primeira arquitetura considerada, denominada arquitetura heterogênea, é utilizada, por exemplo, no Rebeca (Mühl et al., 2004), no Siena (Carzaniga et al., 1998) e no Hermes (Pietzuch & Bacon, 2002), e admite uma diferenciação entre os brokers, isto é, nós da rede responsáveis por disponibilizar as funcionalidades publish-subscribe, e os clientes, isto é, nós que se conectam aos brokers para utilizar a rede publish-subscribe.

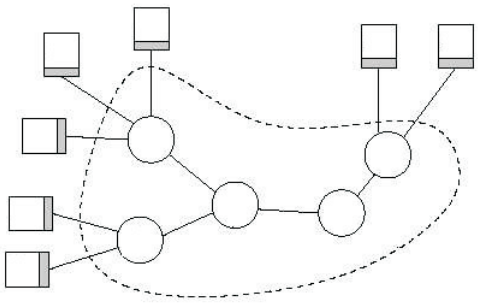

Figura 3.1: Arquitetura Heterogênea

A figura 3.1 (Terpstra et al., 2003) ilustra a arquitetura heterogênea. Cada broker pode atender a mais de um cliente e podem existir brokers que não recebem nenhuma conexão mas que, ainda assim, são necessários para o funcionamento da rede publish-subscribe.

Na arquitetura heterogênea, o código executado nos clientes é diferente do código executado nos brokers. Os brokers gerenciam os tópicos e disponibilizam funções publish-subscribe, enquanto que, aos clientes, cabe apenas executar a aplicação que utiliza as funções disponibilizadas pelos brokers.

Se analisarmos a arquitetura heterogênea, podemos perceber sua proximidade com sistemas que utilizam brokers fixos de alto poder computacional e clientes possivelmente móveis e menos poderosos. Se considerarmos que o poder de processamento e armazenamento dos clientes é menor que o dos brokers, a arquitetura se torna interessante pois o código sendo executado nos clientes pode ser menos custoso. A existência de máquinas fixas com maior poder de processamento sendo disponibilizadas independentemente dos clientes, portanto, pode ser bastante atrativa.

A mobilidade dos clientes também pode se adequar bastante à arquitetura heterogênea. A conexão e desconexão de clientes móveis pode ser bastante freqüente, mas este tipo de re-configuração não requer grandes alterações na estrutura da rede publish-subscribe. A substituição de um broker responsável por gerenciar os tópicos, por outro lado, será significativamente mais custosa. Devemos considerar, entretanto, que uma rede de servidores fixos não deve sofrer alterações freqüentes.

A arquitetura heterogênea, entretanto, possui problemas que devem ser considerados. A principal dificuldade desta arquitetura é o projeto e manutenção de uma rede de servidores fixos funcionando como brokers. A manutenção de um número excessivo de servidores seria custosa. Caso o número de servidores fosse inferior ao necessário, entretanto, a rede ficaria sobrecarregada e o sistema não funcionaria com sucesso. O bom funcionamento da rede está relacionado também com a distribuição dos clientes em relação aos brokers. Brokers que recebam conexões de muitos clientes também têm que realizar um processamento maior, possivelmente ultrapassando sua capacidade e prejudicando a utilização do sistema.

A segunda arquitetura analisada, denominada homogênea, é utilizada, por exemplo, no Scribe (Rowstron et al., 2001) e no Bayeux (Zhuang et al., 2001), e não considera a diferenciação ente clientes e brokers. Nesta arquitetura, todos os nós do sistema têm o mesmo papel. Desta forma, todos os nós estão aptos a gerenciar tópicos, disponibilizar funções publish-subscribe e implementar a aplicação que utiliza as funcionalidades.

A figura 3.2 ilustra a arquitetura homogênea. Todos os nós são idênticos e se comunicam de acordo com a topologia estabelecida. Desta forma, os nós são responsáveis pela manutenção da rede publish-subscribe.

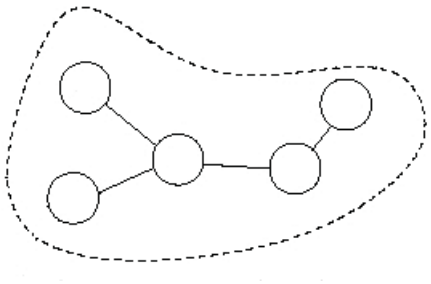

Figura 3.2: Arquitetura Homogênea

A arquitetura homogênea admite que os nós têm um poder de processamento razoável e uma capacidade de armazenamento que permita a gerência de tópicos quando necessário, embora estas premissas possam ser contornadas na camada de aplicação. Caso a aplicação publish-subscribe desenvolvida deva ser utilizada em clientes com recursos escassos, pode-se utilizar uma máquina mais poderosa como gateway para a rede publish-subscribe. Desta forma, os clientes se conectariam ao gateway e este seria responsável pela troca de informações com a rede publish-subscribe. Se considerarmos esta modelagem, a arquitetura de nós idênticos pode lembrar muito a arquitetura heterogênea. Existe, entretanto, uma diferença conceitual fundamental. Na arquitetura de brokers, os clientes executam o código da aplicação e os brokers formam a rede publish-subscribe. Na arquitetura de nós idênticos, os nós também executam o código da aplicação. Conceitualmente, a comunicação entre os clientes e o gateway, que se dará apenas na camada de aplicação, será totalmente independente da rede publish-subscribe.

A mobilidade dos clientes é uma questão tratada com mais transparência nesta arquitetura. Ao contrário da arquitetura heterogênea, a arquitetura homogênea não considera a existência de uma rede publish-subscribe fixa esperando por conexões. A entrada e saída de clientes da rede pode ser feita a qualquer momento, aumentando ou diminuindo o número de máquinas envolvidas. Nestas situações, caberá à própria rede publish-subscribe a garantia de consistência das informações. Desta forma, não há a necessidade de manutenção de uma infra-estrutura fixa de servidores.

A escolha por uma das arquiteturas apresentadas não consiste em determinar a melhor arquitetura, e sim a mais adequada para cada situação. Para o nosso sistema, julgamos que a manutenção de uma rede fixa de servidores afastaria o estudo de seus objetivos. A análise de uma rede publish-subscribe

onde todos os nós são iguais e têm a responsabilidade de gerenciar os tópicos, por outro lado, se aproximaria bastante de uma rede peer-to-peer, compatível com as idéias do protocolo Pastry. Nesta arquitetura, cada nó terá um identificador na rede peer-to-peer e os tópicos serão gerenciados pelos nós cujos identificadores mais se aproximarem do hash do tópico. Para executarmos as funcionalidades publish-subscribe, utilizaremos o hash do nome do tópico e o protocolo pastry para encaminhar as solicitações para o nó adequado.

3.2 Divisão em Camadas

Sistemas publish-subscribe complexos são definidos por diversas decisões de projeto que levam a implementações totalmente diferentes. Podemos diferenciar os sistemas existentes levando em conta, por exemplo, o ambiente de execução, a forma de comunicação ou os serviços disponibilizados. Na maioria dos sistemas existentes, entretanto, as diferentes decisões de projeto não são modularizadas, sendo o sistema final definido pelo conjunto de decisões tomadas. O sistema LuaPS busca se diferenciar destes sistemas monolíticos, definindo diferentes camadas e módulos que agrupam decisões de implementação relacionadas e individualizam as diferentes áreas do sistema.

Todas as decisões de projeto que se relacionam foram agrupadas em uma mesma camada, evitando que decisões ortogonais fossem agrupadas em um mesmo módulo. Cada uma das camadas, embora independente das outras camadas, deve ser capaz de interagir com as camadas vizinhas através de funções que respeitam a interface definida. Desta forma, cada uma das camadas do sistema implementa uma parte da solução final.

A arquitetura de desenvolvimento dividida em camadas também tem sido utilizada em outras áreas de estudo. O modelo de referência OSI (ISO/IEC 7498-1:1994), definido pela ISO (International Organization for Standardization), por exemplo, define uma arquitetura em camadas com a finalidade de padronizar o desenvolvimento de protocolos para redes de comunicação.

O sistema desenvolvido utiliza uma tabela hash distribuída como substrato e é executado em um ambiente orientado a eventos. A separação em camadas nos permite individualizar cada uma das decisões tomadas e, desta forma, entender melhor a solução desenvolvida e perceber as alterações que seriam necessárias para alterar a estrutura do sistema. Desta forma, a arquitetura em camadas nos permite um entendimento mais claro do sistema e dos pontos que cada uma das decisões do projeto envolvem.

A divisão em camadas influi positivamente no processo de desenvolvimento também. Se considerarmos que cada uma das camadas é especializada em uma parte da solução completa, podemos perceber que o desenvolvimento estará mais focado e as soluções implementadas poderão ser mais bem avaliadas. Cada uma das camadas poderá ser desenvolvida e testada separadamente, garantindo que os problemas sejam mais facilmente detectados e corrigidos.

No que se refere ao desenvolvimento, entretanto, a principal contribuição da arquitetura em camadas é a evolução e alteração do sistema. Uma vez que a interface continue sendo respeitada, melhorias nas funcionalidades podem ser feitas sem que as outras camadas do sistema sejam afetadas. Poderíamos, na verdade, substituir toda uma camada, possivelmente por uma camada com uma implementação totalmente nova, que o sistema continuaria consistente desde que as interfaces fossem respeitadas.

A utilização de diferentes camadas, entretanto, também possui aspectos negativos. A principal desvantagem da utilização desta arquitetura é a necessidade de uma avaliação mais estruturada da solução, buscando determinar quais problemas devem ser tratados em cada camada. Desta forma, o processo inicial de desenvolvimento pode ser mais lento.

Uma outra dificuldade introduzida pela utilização de camadas no desenvolvimento do sistema se relaciona ao acesso às estruturas internas das camadas. Para que a modularização em camadas seja respeitada, cada uma das camadas só deve ser acessada através das funções disponibilizadas pela interface. Desta forma, quando o acesso externo a estruturas internas for interessante, deve-se pensar em funções que devem ser disponibilizadas na interface.

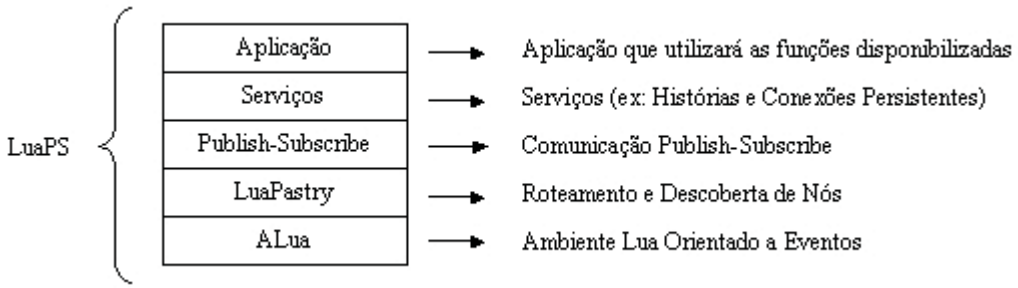

Figura 3.3: Camadas do Projeto LuaPS

Após analisarmos os benefícios e dificuldades introduzidas pela utilização da arquitetura em camadas, consideramos que os benefícios justificam a sua utilização. A figura 3.3 ilustra a arquitetura em camadas implementada pelos nós da rede, especificando a função de cada uma das camadas.

As duas camadas inferiores do sistema, a camada ALua e a camada LuaPastry, utilizam sistemas já existentes e que já foram apresentados. A camada ALua disponibiliza um ambiente orientado a eventos utilizando a linguagem de programação Lua e a camada Pastry provê uma infra-estrutura de rede baseada em Tabela Hash Distribuída.

3.2.1 Camada Publish-Subscribe

A camada publish-subscribe é a primeira camada que foi desenvolvida especificamente para o projeto. Esta camada disponibiliza funções para entrada e saída da rede publish-subscribe, criação de tópicos, inscrição e remoção de inscrições em tópicos e publicação de notificações. Para garantir a execução correta das funcionalidades citadas, a camada publish-subscribe gerencia os tópicos e as inscrições, implementando estruturas particulares que não são acessadas por outras camadas. Para garantir a generalidade do sistema, a camada publish-subscribe utiliza uma função de publicação genérica, definida pelas camadas superiores do sistema.

A camada publish-subscribe é responsável, ainda, pela manutenção do estado consistente das informações em qualquer instante, mesmo considerando a entrada e saída de nós da rede a qualquer momento. Para garantir a execução das funções de forma consistente, a camada publish-subscribe implementa uma função para a gerência da execução dos comandos. Ainda considerando a extensão do sistema com o desenvolvimento de uma camada de serviços, a camada publish-subscribe disponibiliza funções que permitem associar novos campos à estrutura de gerência dos tópicos. Desta forma, a camada assume a responsabilidade de realocar as informações associadas aos tópicos quando ocorrem entradas ou saídas de nós da rede.

3.2.2 Camada de Serviços

A camada de serviços tem como principal objetivo estender a camada publish-subscribe com funcionalidades que não caracterizam o paradigma de comunicação publish-subscribe, mas que podem ser necessários para a aplicação publish-subscribe sendo desenvolvida.

Se estendermos a discussão sobre individualização de camadas no sistema LuaPS para a camada de serviços, podemos considerar que serviços diferentes

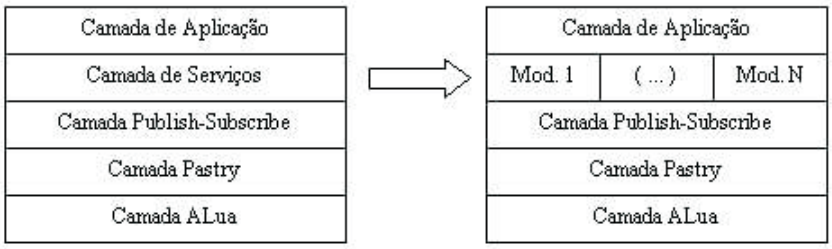

Figura 3.4: Módulos da Camada de Serviços

devam ser individualizados em módulos particulares. Desta forma, podemos considerar que a camada de serviços será composta por um ou mais módulos que irão disponibilizar os serviços. Cada um dos módulos poderá ser completamente independente dos outros. A figura 3.4 ilustra a arquitetura da camada de serviços.

No projeto LuaPS desenvolvido, implementamos três módulos de serviço que irão complementar o sistema. Embora estes serviços sejam interessantes para um sistema publish-subscribe, a principal função desta implementação é servir como modelo para a implementação de outros módulos de serviço e evidenciar a extensibilidade da arquitetura.

Sistemas publish-subscribe tradicionais não disponibilizam acesso a notificações antigas. Isto é, uma notificação é garantidamente entregue a um cliente se e somente se o mesmo tiver se inscrito no respectivo tópico previamente. Para muitas aplicações, entretanto, este tipo de comportamento pode não ser o mais adequado. O módulo *luapsh* desenvolvido se baseou no conceito de histórias para possibilitar o acesso a notificações antigas. O módulo disponibiliza funções para gerar o histórico de publicações e acessar as informações armazenadas. Podemos encontrar uma discussão sobre o acesso a notificações antigas em (Cilia et al., 2003) ou analisando o Rebeca (Mühl et al., 2004), um sistema que utiliza o conceito de histórias.

Uma outra questão bastante abordada em sistemas publish-subscribe é a utilização de clientes móveis. A possível instabilidade na conexão e limitação de recursos dos dispositivos nos indica a necessidade de tratamento especial. O módulo *luapsg* desenvolvido implementa um gateway publish-subscribe para clientes móveis com conexões persistentes. Com a utilização deste módulo, clientes móveis podem executar uma versão simplificada da aplicação, mais adequada aos recursos disponíveis. A existência de conexões persistentes, por outro lado, garante que as mensagens enviadas para os clientes não serão perdidas, mesmo em momentos de desconexão. Desta forma, o módulo garante a consistência das informações mesmo para dispositivos com conexões instáveis.

O terceiro módulo disponibilizado busca prover tolerância a falhas. Com a arquitetura utilizada, os clientes da rede publish-subscribe são responsáveis por gerenciar as informações referentes aos tópicos criados e inscrições realizadas. A falha de um nó, portanto, pode acarretar na perda de informações. O módulo *luapsb* desenvolvido utiliza os nós vizinhos para realizar backup das informações, possibilitando que a rede se re-estruture em caso de falha.

3.2.3 Camada de Aplicação

O desenvolvimento de uma camada de aplicação não é um dos pontos mais importantes do projeto, devendo ficar sob responsabilidade dos desenvolvedores que vierem a utilizar o sistema publish-subscribe. O desenvolvimento da camada de aplicação, entretanto, se faz necessário para exemplificar o processo, possibilitar a realização de testes completos e ilustrar as situações descritas.

Para realizar os testes e demonstrações necessárias, desenvolvemos diferentes módulos para a camada de aplicação. Estes módulos são simples mas, integrados com o resto do sistema, nos possibilitam o vislumbre de um sistema publish-subscribe completo.

Segue uma lista com os módulos implementados e uma breve descrição:

- testeTimer - Testas as funções publish-subscribe e o módulo de histórias.
- testeConsole - Testa o console de interação com o usuário.
- testeGateway e testeGatewayApp - Testa o módulo de gateway.
- testeBackup - Testa o módulo de tolerância a falhas.
- testeCarga - Realiza um teste de carga no sistema.