4
Estudo de Casos

Para verificar a flexibilidade do framework proposto, escolhemos dois aplicativos encontrados na literatura para serem reimplementados a partir dele: *Nita* e *BuddySpace.*

Nita foi selecionado para servir como estudo de caso porque apresenta uma série de características que são comuns a várias ACBLs. Além disso, provê tanto comunicação síncrona quanto assíncrona, ao contrário de outras aplicações que provêem apenas um desses dois modos de comunicação.

Quanto ao *BuddySpace*, sua escolha justifica-se pelo fato de ser a aplicação que mais se difere das outras apresentadas nos trabalhos relacionados, pois não segue o paradigma de comunicação através da postagem de mensagens em localidades. Ao invés disso, a localização é utilizada para enriquecer a informação de contexto compartilhada entre os usuários. Além disso, os dois aplicativos diferem entre si em vários aspectos. Por exemplo, *BuddySpace* utiliza sockets TCP para comunicação e posicionamento manual. *Nita*, por sua vez, usa um *middleware publish/subscribe* e tecnologia de localização baseada na rede. Essas diferenças dão uma boa idéia da flexibilidade do framework, e ilustram como a instanciação dos pontos adaptáveis permite atender a diferentes requisitos.

4.1.
Nita

Nita faz parte de uma classe de *ACBLs* que têm em comum prover comunicação entre usuários de dispositivos móveis através do envio de mensagens a localidades. Além dessa forma assíncrona de comunicação, *Nita* ainda oferece suporte à comunicação síncrona. Com sua instanciação objetivamos mostrar que o framework é flexível o bastante para criar aplicações que utilizam essas duas formas de comunicação.

4.1.1. O Servidor

Em sua essência, o servidor *Nita* serve como difusor de eventos, gerenciador de usuários, repositório de dados e intermediário na comunicação entre os diversos clientes. Para sua instanciação, vamos analisar as implementações padrões dos pontos adaptáveis fornecidas pelo framework e verificar se satisfazem os requisitos da aplicação.

4.1.1.1. Tipo de Comunicação

O cliente e o servidor *Nita* comunicam-se através de mensagens contendo comandos. Por exemplo, o cliente envia comandos para registrar um novo usuário, postar mensagens em uma localidade, etc. Por sua vez, o servidor envia ao cliente informações sobre mudanças na sua localização, mensagens postadas e de erro, etc.

A parte de comunicação do framework possui três pontos adaptáveis: *CommunicationIF, CommunicationFactoryIF* e *RouterIF*. O primeiro é a classe que encapsula a forma de comunicação, o segundo é a fábrica que cria objetos para essa forma de comunicação e o último é a classe que vai receber os comandos que chegam da rede através de *CommunicationIF* e roteá-los para outras classes do sistema.

Nita utiliza o serviço de eventos *ECI* da arquitetura *MoCA* para comunicação entre o cliente e o servidor, que possibilita uma comunicação segundo o paradigma *publish/subribe,* baseada em tópicos. O servidor registra-se como assinante no tópico "server" para receber mensagens dos clientes, e publica mensagens no tópico de cada cliente. Os clientes, por sua vez, registram-se como assinantes no tópico que tem como nome o *login* do seu usuário e publicam mensagens no tópico "server".

O framework já oferece a classe *MocaPubSub,* uma implementação de *CommunicationIF* para *ECI,* e *CommunicationFactory,* uma implementação de *CommunicationFactoryIF* que, além de objetos *TcpSocket,* cria também objetos *MocaPubSub*. Assim, na instanciação do *Nita* não foi necessário implementar essas interfaces.

A interface *RouterIF* foi implementada pela classe *NitaServer,* que funciona como classe principal do *Nita*, responsável pela autenticação do usuário e instanciação da aplicação. Implementando *RouterIF* e seu método *onCommand*, *NitaServer* passa a ficar responsável também por receber os comandos de *MocaPubSub,* verificar seus tipos e roteá-los para os objetos que irão tratá-los.

4.1.1.2. Informação de Localização

Com relação à localização, o framework oferece dois pontos adaptáveis: *PositioningAdapterIF* e *SymbolicRegionManagerIF*. O primeiro trata da tecnologia de localização a ser utilizada e o segundo é responsável pelo mapeamento de regiões físicas em lógicas.

Nita não usa informação manual de localização, onde o usuário informa sua posição, mas sim detecção de localização centrada na rede, onde a localização do usuário é inferida de forma automática. Para isso, *Nita* usa um outro serviço fornecido pelo *middleware MoCA,* o *Location Inference Service (LIS).*

Como o framework já oferece a classe *LisAdapter,* uma implementação da interface *PositioningAdapterIF* para o *LIS*, *Nita* simplesmente a reutiliza. Para ilustrar como é feita essa instanciação de *PositioningAdapterIF*, a seguir descrevemos como *LisAdapter* foi criada.

No construtor dessa classe, obtêm-se as informações de rede (IP, porta) do *host* que executa o *LIS*. Este serviço necessita do *MAC address* do dispositivo móvel para informar a sua localização. Assim, em *registerUser*, deve ser passado o login do usuário e, como objeto de configuração, o *MAC address*, ou seja, o endereço da placa de rede do dispositivo móvel. Com essas informações, *LisAdapter* registra-se junto ao *LIS* para ser notificado quando esse dispositivo mudar de localização. Quando isso ocorrer, *LisAdapter* chama o método *setUserLocation* em *Postioning.*

LIS permite que a informação da localização de um usuário seja obtida também por consulta direta, e não apenas de forma assíncrona, por notificações. Assim, o método *getPosition* de *PositioningAdapterIF* foi implementado, passando-se o *MAC address* do dispositivo móvel como parâmetro. Com isso, a implementação de *LisAdapter* está concluída.

A classe responsável por mapear regiões físicas em simbólicas deve implementar a interface *SymbolicRegionManagerIF*. *LIS,* além de ser um serviço de inferência de localização, também é o responsável pela gerência de regiões simbólicas aninhadas, que formam uma hierarquia de regiões. Assim, o framework oferece também a classe *LisRegionManager(LRM),* que implementa *SymbolicRegionManagerIF* e serve como fachada para esse serviço.

Como dito anteriormente, *LIS* já informa a localização simbólica de um usuário e não apenas a sua posição. Assim, não há necessidade de se fazer uma tradução desse dado. Dessa forma, *LRM* implementa o método *translate* apenas retornando o mesmo parâmetro que lhe foi passado. Já na implementação do método *getSymbolicRegions*, *LRM* consulta o *LIS* e retorna todas as regiões simbólicas mapeadas.

Como as implementações padrões do framework relativas à obtenção da informação de localização satisfizeram os requisitos de *Nita,* não foi necessário estender esse ponto adaptável.

4.1.1.3. Localidades

No pacote *flocs.location,* existem dois *hot spots* principais: *Location,* que é a representação no sistema de uma localidade, e *Message*, que são representa as mensagens a serem postadas em uma localidade.

Nita permite que usuários enviem (postem) mensagens nas localidades por onde passam. Toda mensagem tem um tempo de validade, e as localidades devem excluir mensagens quando estas estiverem expiradas. As classes *Message* e *Location*, oferecidas pelo framework, possuem essas funcionalidades.

Na instanciação do *Nita, Location* foi utilizada sem necessidade de nenhuma modificação. Quanto à *Message*, por ser uma classe abstrata, essa precisou ser estendida por uma subclasse chamada *Note,* que implementou seu método abstrato *getMessageType*. Na sua versão atual, *Nita* trabalha apenas com mensagens do tipo texto. Sendo assim, esse método de *Note* retorna como tipo de mensagem a constante string *Message.TEXT*.

As mensagens usadas para comunicação síncrona têm seu atributo *expirationMode* ajustado para o valor *INSTANTANEOUS,* para que não fiquem

guardadas na localidade. Elas são apenas repassadas para os usuários que se encontram na mesma localidade, no instante em que foram postadas. Já para a comunicação assíncrona, *expirationMode* é igual a *EXPIRABLE*, indicando que a mensagem possui um tempo de permanência na localidade, tempo este (de expiração) determinado pelo usuário.

4.1.1.4. Persistência de Dados

DataServiceIF é a interface que permite implementar persistência de dados.

Nita necessita de uma forma de persistir os perfis dos usuários, as configurações das localidades e as mensagens postadas nestas localidades. Como a classe *FileService* do framework, que implementa *DataServiceIF,* executa essas funções, não foi necessário implementar um novo ponto adaptável para a base de dados.

4.1.1.5. Usuários e registro em eventos

No pacote *flocs.auxiliar* existem dois pontos adaptáveis: a classe abstrata *User,* que representa o usuário no sistema, e a classe *Profile,* representando o perfil do usuário.

Na versão original do *Nita,* a classe *NitaServer* centralizava o recebimento de todos os eventos do sistema (postagem de uma mensagem, mudança de localização, etc) e se incumbia de repassá-los para os clientes interessados. As informações dos usuários da aplicação cliente eram guardadas numa classe similar à *User*.

Por outro lado, na instanciação do *Nita,* optou-se por uma redistribuição de obrigações. Ao invés da centralização em *NitaServer,* cada objeto *User* deve se registrar nos eventos de interesse do usuário que representa e se encarregar das notificações. Foi criada, então, a classe *NitaUser* como subclasse de *User,* que herda os atributos desta e acrescenta novos métodos implementados das interfaces *MessageInLocationListenerIF* e *PositionListenerIF*.

A primeira interface permite que *NitaUser* registre-se na localidade corrente de seu usuário para que este seja notificado quando uma mensagem for postada.

Para isso, foi preciso implementar os métodos *messagePosted* e *messageRemoved* dessa interface. Quando o primeiro é chamado, *NitaUser* envia a mensagem passada como parâmetro para a aplicação cliente no dispositivo móvel. O segundo não precisou ser implementado, já que *NitaUser* não tem interesse em saber quando uma mensagem for removida.

Já a interface *PositionListenerIF* permite que se registre em *Positioning* para ser notificada quando o usuário que representa mudar sua localização. Para isso, foi necessário implementar o método *locationChanged* dessa interface. Quando este método é chamado, *NitaUser* remove seu usuário da localidade anterior e associa-o à nova localidade passada como parâmetro. Além disso, adiciona-se como um *MessageInLocationListener* dessa nova localidade. Depois, obtém as mensagens que se encontram na nova localidade e as envia para o cliente através de *CommunicationService.*

A classe *Profile* não precisou ser modificada. *NitaUser* a utiliza para guardar a informação de quais mensagens já foram lidas. Depois que uma mensagem for entregue para o usuário, *NitaUser* chama o método *setRead* de *Profile* passando-a como parâmetro, para que a mesma seja colocada na lista de mensagens lidas. E sempre quando um usuário entra em uma nova localidade, antes que as mensagens postadas nesta localidade sejam-lhe enviadas, *NitaUser* invoca o método *getNotRead* passando a lista de mensagens já lidas como parâmetro e recebendo uma lista das mensagens novas não lidas, que efetivamente deverão ser encaminhadas para o usuário.

4.1.1.6. Registro dos componentes

Quando todos os pontos adaptáveis do framework tiverem sido implementados, é necessário então registrá-los na classe *ComponentRegistry(CR),* do servidor. Então, na classe principal do sistema, que no caso do aplicativo *Nita* é o *NitaServer*, deve-se obter uma referência à instância única de *CR,* chamando o método estático *getInstance* dessa classe e, através dos diferentes métodos *set*, registrar todos os pontos adaptáveis modificados. Abaixo, um pequeno trecho do código do construtor de *NitaServer* que executa essa tarefa:

```
...

//obtem a instancia de ComponentRegistry
ComponentRegistry cr = ComponentRegistry.getInstance();

//Ponto adaptável de Comunicação
cr.setRouter(this);

//Pontos adaptáveis de Tecnologia de Localização
cr.setPositioningAdapter(new LisAdapter());
cr.setSymbolicRegionManager(new LisRegionManager());

...
```

Como na parte de comunicação foram reutilizadas todas as classes, com exceção de *RouterIF,* que foi implementada por *NitaServer*, apenas a implementação desta última teve que ser registrada em *CR.*. Quanto à tecnologia de localização, *LisAdapter* e *LisRegionManager* também tiverem de ser registradas em *CR*, apesar de *Nita* as ter reutilizado. Isso porque a classe *ManualPositioning,* que implementa *PositioningAdapterIF* usando posicionamento do tipo manual (informado pelo usuário), é a classe padrão registrada em *CR.*

4.1.1.7. Comandos e Fluxo de Comandos

O framework oferece várias classes que representam mensagens/comandos cujos atributos são usados para a troca de informação entre o cliente e o servidor. Como o framework foi baseado na experiência com o desenvolvimento do protótipo *Nita*, grande parte dos comandos oferecidos pelo framework foi criada de acordo com os requisitos deste. Assim, essas classes foram integralmente reutilizadas na sua instanciação.

O fluxo dos comandos que chegam ao servidor é definido pela classe *NitaServer*, que implementa a interface *RouterIF*. Um pequeno trecho do método *onCommand* é apresentada a seguir:

```
public void onCommand(Object obj) {
       if (obj instanceof Login)
           login((Login)obj);
       else if (obj instanceof Logout)
           logout((Logout)obj);
       else if(obj instanceof CreateUser)
           createUser((CreateUser)obj);
       else if (obj instanceof MessageCommand){
           MessageCommand cmd = (MessageCommand)obj;
           _locationMng.post(cmd.message, cmd.location);
       }
```

```
                    ... (continua)
    }
```

Primeiro, é verificado o tipo de comando e depois este é repassado para o método da classe que deve tratá-lo. No exemplo, os três primeiros comandos (login, logout e criação de usuário) são passados para métodos da própria classes *NitaServer*. Já *MessageCommand,* que representa uma mensagem postada por um usuário em uma localidade, tem seus atributos passados diretamente para o método *post* de *LocationManager*.

A definição de quais comandos devem ser criados e como será o fluxo deles no sistema deve partir da análise dos casos de uso do sistema. Os casos de uso principais do *Nita* são os mesmo do framework, que foram descritos na Seção 3.3.7. Neste caso, basta substituir os pontos adaptáveis do framework pelas classes que os implementaram na instanciação do *Nita*. Por esse motivo, não os descrevemos novamente nesta seção.

4.1.1.8. Execução do Servidor

Após a instanciação dos pontos adaptáveis e a definição do fluxo de comandos no sistema, o servidor pode ser executado. Para isso, deve-se iniciar o serviço de comunicação da aplicação instanciada. O trecho de código abaixo, retirado da classe *NitaServer,* mostra como isso é feito:

```
                    ...

ComponentRegistry cr = ComponentRegistry.getInstance();
_communication = cr.getCommunicationService();
_communication.initServer(PORT);

                    ...
```

Primeiro, obtém-se uma referência de *CommunicationService* a partir de *ComponentRegistry*. Depois, chama-se seu método *init,* passando como argumento o número da porta em que a *thread* servidora de comunicação vai fazer o *bind.* Feito isso, o servidor está pronto para receber comandos dos clientes através da rede.

A Figura 8 mostra o diagrama de classes do servidor *Nita.*

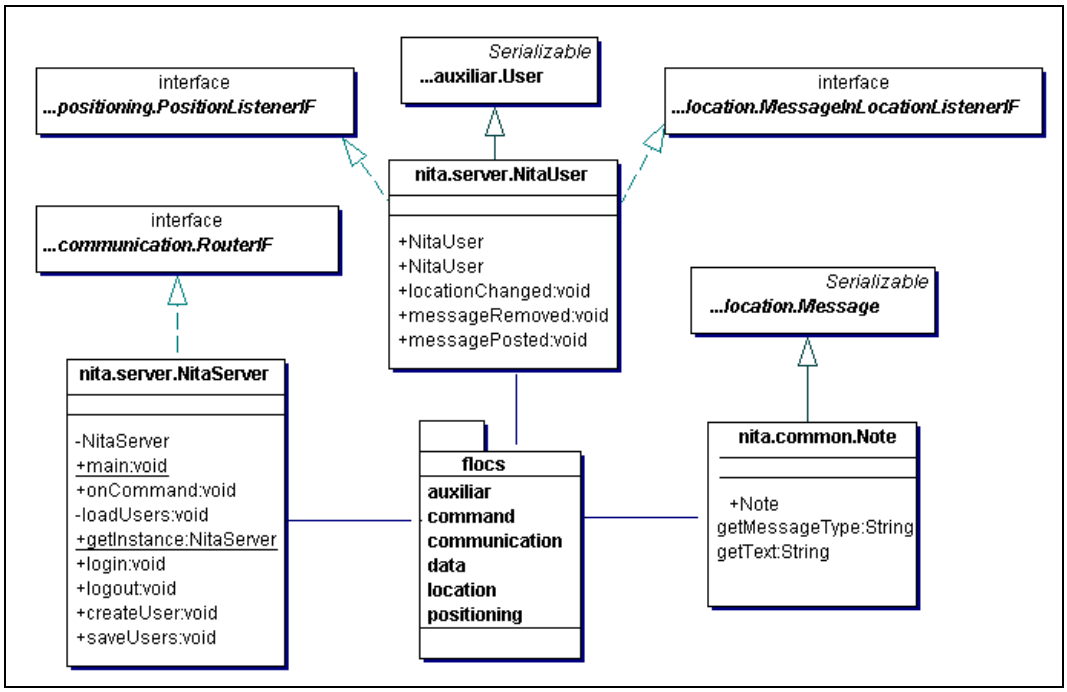

Figura 8 - Diagrama de classes simplificado do servidor Nita.

4.1.2. O Cliente

O cliente *NITA* foi desenvolvido em J2SE e pode ser utilizado em notebooks dotados com interface de rede sem fio. Futuras versões poderão ser implementadas em J2ME para que o programa possa ser executado em outros dispositivos computacionais móveis mais limitados, como palmtops, handhelds, smart-phones, celulares, etc.

4.1.2.1. Arquitetura

No desenvolvimento do cliente da aplicação, foram reutilizados os serviços de comunicação e persistência de dados e as classes de comandos oferecidos pelo framework. Assim, foi necessário implementar apenas a interface gráfica com o usuário e a lógica de acesso a esses serviços.

A classe principal é *NitaClient,* que comunica-se com todos os outros componentes. Como apenas uma única instância dessa classe deve existir no sistema, ela foi criada com o padrão de projeto *Singleton*. *NitaClient* implementa a interface *RouterIF* para receber os comandos do servidor que chegam pela rede

através de *MocaPubSub* e utiliza *CommunicationService* para enviá-los para o servidor *Nita*. Além disso, dá suporte à persistência dos dados do usuário e das mensagens a serem salvas através do uso do componente *FileService.*.

NitaClient também é responsável por controlar a interface gráfica com o usuário. O cliente foi desenvolvido de forma a ser independente de interface gráfica. Para isso, foi criada a interface *NitaUI*, contendo métodos que devem obrigatoriamente ser implementados pela classe de interface com o usuário[6].

A Figura 9 mostra o diagrama de classes simplificado do cliente *Nita.*

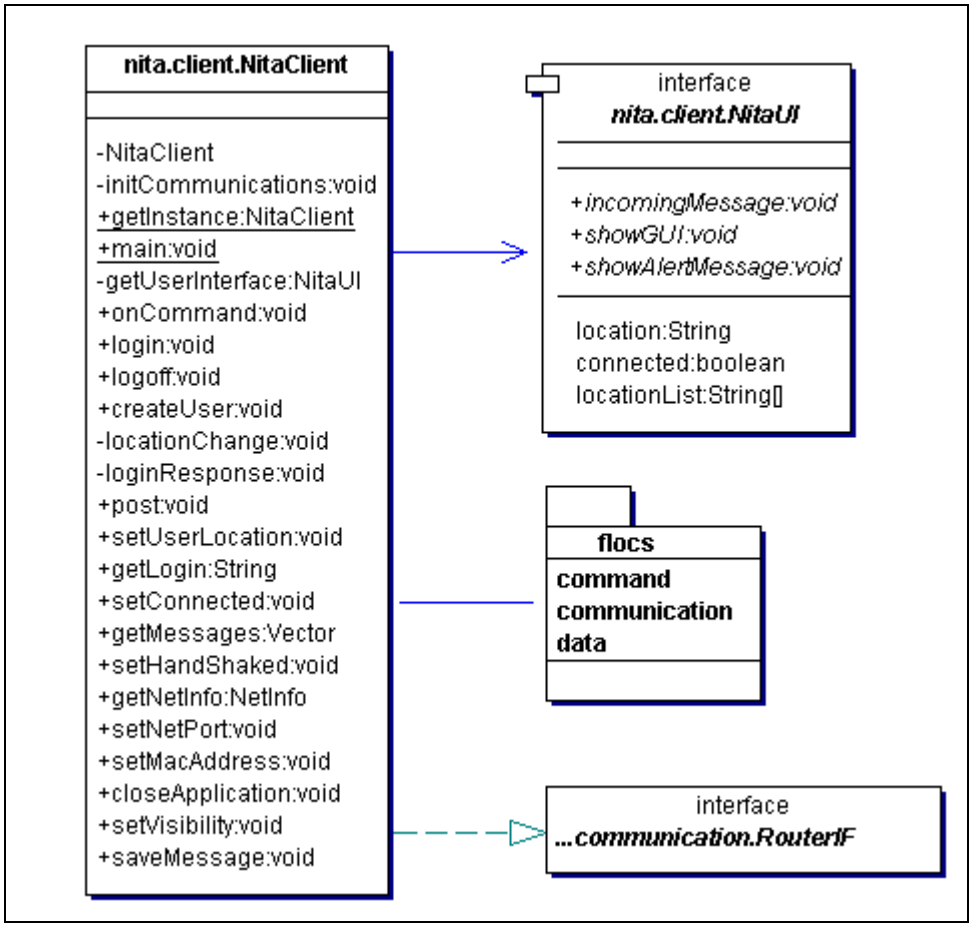

Figura 9 - Diagrama de classes simplificado do cliente *Nita.*

[6] A interface gráfica do *Nita* foi projetada e desenvolvida por Bruno Santana da Silva, do SERG/PUC-Rio.

4.1.2.2. Exemplo de cenário

Quando o usuário inicializa a aplicação *Nita*, uma janela aparece para ele faça o login e forneça a senha para acessar o servidor *Nita* (Figura 10).

Caso o login seja um bem sucedido, a janela principal do aplicativo é mostrada, informando na parte inferior a localidade corrente do usuário (no caso, o Laboratório de Engenharia de Software/LES) (Figura 11).

Para enviar uma mensagem, o usuário deve clicar em *"Nova Mensagem"*. Então, a janela da Figura 12 é mostrada, onde o usuário pode escrever sua mensagem, escolher na árvore representando a hierarquia de regiões simbólicas o local de destino da mensagem, bem como sua data de expiração ("limite de entrega"). O destinatário da mensagem também pode ser especificado, sendo possível que seja um indivíduo em particular, ou um conjunto de indivíduos, reunidos em uma *buddylist*, ou todos aqueles que estão naquela localidade. Na figura, por exemplo, Maria posta uma mensagem para todos os usuários que se encontram no LAC.

Para ler as mensagens postadas na sua localidade corrente, o usuário clica em *"Ler Mensagem"* na janela principal. A janela da Figura 13 aparece, mostrando uma árvore de hierarquia com a localidade corrente representada por um nó selecionado e, ao lado, o cabeçalho das mensagens que foram postadas naquelas localidades. Ao clicar sobre uma mensagem, o seu conteúdo é mostrado na parte inferior da janela.

Figura 10 - Tela de Login do *Nita*.

Figura 11 - Janela principal do *Nita*.

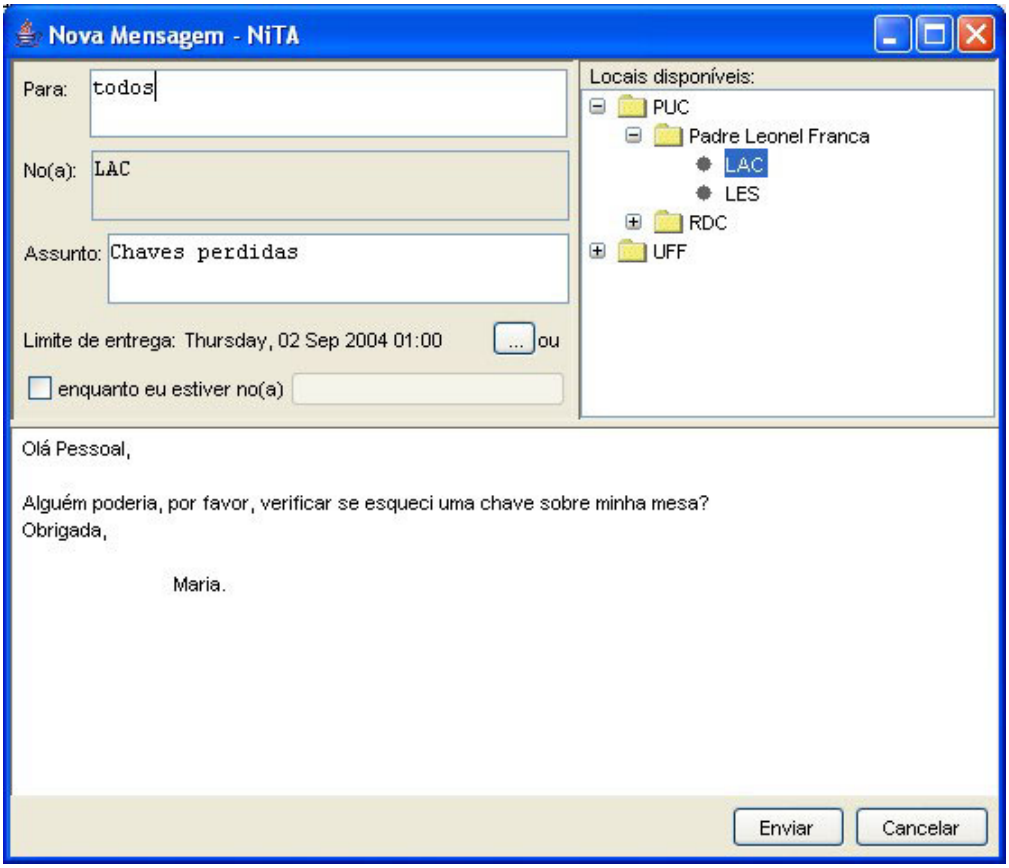

Figura 12 - Janela de envio de mensagem.

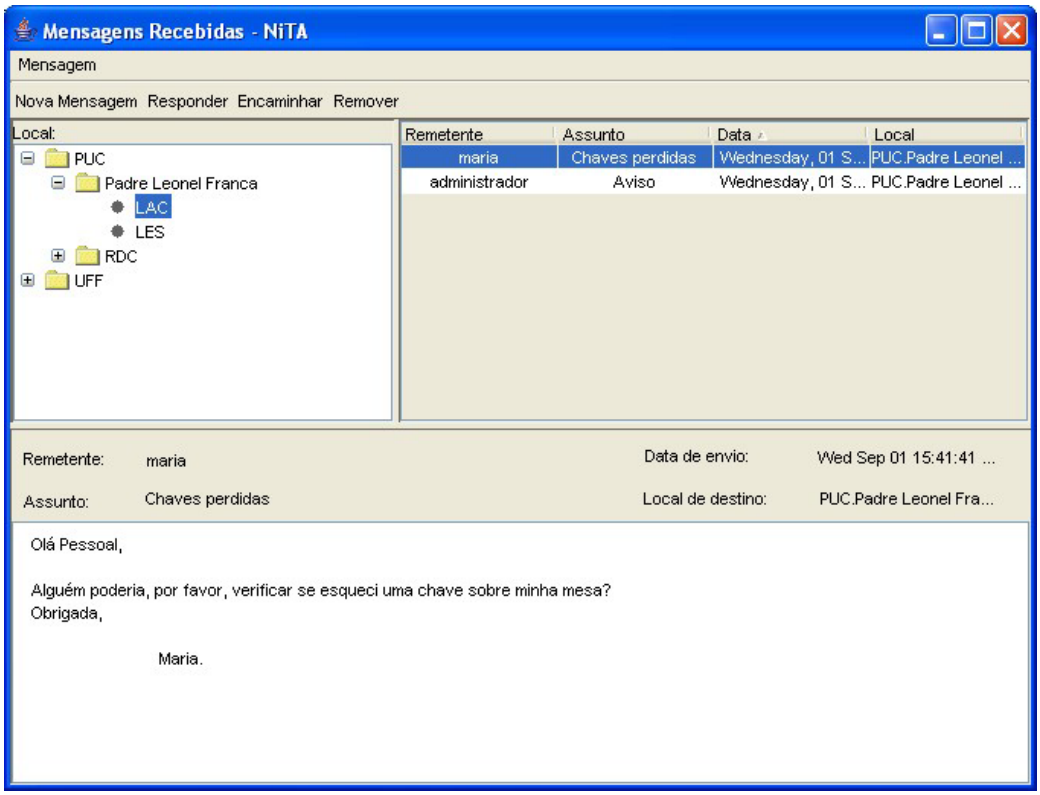

Figura 13 - Janela de recebimento de mensagem.

4.1.3. Conclusões

A Tabela 4 mostra a reutilização de classes na instanciação do *Nita*. A primeira coluna apresenta os pacotes do framework e onde esses foram utilizados, se no servidor e/ou no cliente da aplicação. A segunda coluna especifica as interfaces que foram implementadas juntamente com as classes que a implementaram. Da mesma forma, a seguir são mostradas as classes seguidas das subclasses que a estenderam. Por fim, são listadas as classes que foram totalmente reutilizadas.

Observando a Tabela 4 nota-se que houve um alto grau de reutilização de código. De fato, no servidor, apenas *NitaServer* teve que ser efetivamente criada. *NitaUser* estendeu *User* apenas para tratar os eventos das interfaces *PositionListenerIF* e *MessageInLocationListenerIF*. Todas as demais classes foram totalmente reutilizadas. No cliente, *NitaClient* serve como classe controladora, criando comandos a partir dos dados entrados pelo usuário através da interface gráfica e repassando-os ao serviço de comunicação do framework

para ser enviado ao servidor. Na instanciação do cliente, uma boa parcela da complexidade está na implementação da interface gráfica com o usuário.

Pacote	Interfaces Implementadas	Classes Estendidas	Classes Reutilizadas
Communication (Servidor/Cliente)	RouterIF (NitaServer/NitaClient)		CommunicationService, CommunicationFactory, MocaPubSub
Positioning (Servidor)	PositionListenerIF (NitaUser)		Positioning, LisAdapter, LisRegionManager
Location (Servidor)	MessageInLocationListenerIF (NitaUser)	Message (Note)	Location, LocationIO, LocationConf, LocationManager
Data (Servidor/Cliente)			FileIO, FileService
Auxiliar (Servidor)		User (NitaUser)	Profile, ComponentRegistry
Command (Servidor/Cliente)			CreateUser, LocationChange, LocationCommand, Login, Logout, LoginResponse, MessageCommand, SimpleMessage, ErrorMessage

Tabela 4 – Reutilização de classes na instanciação do *Nita*.

Naturalmente, esse alto grau de reutilização em parte reflete o fato de o projeto do framework ter sido baseado na experiência adquirida com o desenvolvimento do protótipo *Nita*. Entretanto, como o *Nita* apresenta características comuns à boa parcela das aplicações que seguem o paradigma de comunicação baseada em localização, acreditamos que um grau de reutilização parecido possa ser alcançado na instanciação de outras aplicações desse tipo.

Essa versão do *Nita* instanciada a partir do *FLoCS* pode ser também considerada um framework. Isso porque nessa instanciação a possibilidade de reutilização de código é levada em conta. Por exemplo, os métodos e atributos de todas as suas classes foram criados como *protected,* para que subclasses tenham acesso a eles. Assim, a partir do *Nita,* podem ser instanciados outros aplicativos, reutilizando algumas funcionalidades implementadas no servidor que são recorrentes em um grande número de aplicações, tais como autenticação, criação e gerenciamento de usuários *online*, notificação de eventos aos usuários e etc. Estendendo suas classes e criando uma nova interface gráfica, funcionalidades

podem ser facilmente adicionadas para criar novas aplicações que seguem o mesmo paradigma, mas que implementam serviços diferentes.

4.2. BuddySpace

BuddySpace (Vogiazou et al., 2003) é um programa de mensagens instantâneas que se diferencia dos outros encontrados no mercado por informar a localização geográfica dos usuários. Ao contrário do *Nita, BuddySpace* não oferece comunicação através de envio de mensagem à localidades, mas usa a informação de localização do usuário como forma de enriquecer e assim estimular a comunicação entre usuários.

BuddySpace é um programa grande e complexo. Sendo assim, na sua instanciação, foi implementado apenas um subconjunto de suas funcionalidades. As duas principais funcionalidades instanciadas foram: visualização da localização de cada um dos usuários cadastrados na sua lista de amigos e envio de mensagens instantâneas. A primeira funcionalidade ilustra como é feito o registro nos eventos emitidos pelo framework (no caso, notificação da mudança de localização de um usuário). A segunda funcionalidade mostra como o framework provê comunicação síncrona. As demais funcionalidades, tais como gerenciamento de grupos e compartilhamento de arquivos, não foram implementadas, pois não são específicas da lógica de uma ACBL.

4.2.1. O Servidor

Da mesma forma que o servidor *Nita*, o servidor *BuddySpace* serve como difusor de eventos, gerenciador de usuários, repositório de dados e intermediário na comunicação entre os diversos clientes.

4.2.1.1. Tipo de Comunicação

A comunicação entre os clientes e o servidor da aplicação é feita com o uso de sockets TCP. Essa funcionalidade é oferecida pelo framework através da classe *TcpSocket,* uma das implementações para *CommunicationIF*. Para criar objetos do

tipo *TcpSocket,* pode-se reutilizar também a classe *CommunicationFactory,* implementação padrão de *CommunicationFactoryIF*.

Por fim, implementou-se também a interface *RouterIF* para analisar os comandos que chegam pela rede e roteá-los para os objetos que vão tratá-los. A classe *BuddySpaceServer* foi criada com esse intuito. Além de executar essa tarefa, também é responsável pela autenticação e criação de novas contas de usuários.

4.2.1.2. Informação de Localização

No *BuddySpace,* a informação de localização de um usuário móvel é obtida de forma manual, isto é, cabe aos usuários indicarem onde se encontram. Assim, no cliente da aplicação foi reutilizada a classe *LocationCommand* para criar um comando que permite ao cliente informar ao servidor a localização do usuário. No servidor, como implementação de *PositioningAdapterIF,* reutilizou-se a classe *ManualPositioning,* oferecida pelo framework, que trata de tecnologias de localização centradas no aparelho *(device-centric)*, como é o caso no *BuddySpace.*

Quando *LocationCommand* chega no servidor, a classe *BuddySpaceServer,* que implementa *RouterIF*, encaminha esse comando diretamente para *Positioning,*chamando seu método *setUserLocation.* Dessa forma, *Positioning* notifica os interessados que esse usuário mudou de localização.

Como gerenciador de regiões simbólicas, criou-se a classe *RegionManager,* subclasse de *SymbolicRegionManagerIF.* Seu método *getSymbolicRegions* retorna um vetor de strings com as regiões simbólicas pré-definidas. Como o próprio cliente já informa a sua localização na forma simbólica e não na forma de coordenadas (“bruta”), o método *translate* não faz nada além de retornar a mesma string recebida como parâmetro.

4.2.1.3. Localidades

BuddySpace necessita de uma classe que guarde a informação de quais usuários encontram-se numa região. Para isso, foi reutilizada a classe *Location*.

Como *BuddySpace* não permite que se postem mensagens em locais, não houve necessidade de se reutilizar a classe *Message*, oferecida pelo framework.

4.2.1.4. Persistência de Dados

Na aplicação *BuddySpace* faz-se necessário persistir nos clientes os perfis dos usuários, juntamente com suas listas de amigos, e as mensagens recebidas. Como a classe *FileService* do framework, que implementa *DataServiceIF,* já executa essas funções, não foi necessário implementar um novo ponto adaptável para a base de dados.

4.2.1.5. Usuários e registro em eventos

Para representar os usuários no sistema foi criada no servidor a classe *Buddy,* subclasse de *User*. Além de guardar informações sobre o usuário que representa, *Buddy* serve como um *proxy*, que registra-se em eventos de interesse do usuário e o notifica quando tais eventos ocorrem.

Quando um usuário é autenticado no sistema, uma classe *Buddy* é criada para representá-lo. Esta se incumbe de registrá-lo em *CommunicationService*, para que possa enviar dados via rede para o cliente móvel, e em *Positioning,* para notificá-lo da mudança de sua localização. Além disso, através da implementação da interface *PositionListenerIF,* registra em *Positioning* o interesse pela mudança de localização dos outros usuários cadastrados em sua lista de amigos. Quando tal evento ocorre, o método *locationChanged* dessa interface é invocado. Então, *Buddy* cria um comando *LocationCommand*, com o login do amigo e a sua nova localização, e o envia para o cliente da aplicação interessado. Este, por sua vez, vai atualizar a posição do ícone que representa tal usuário no mapa da interface gráfica.

Para a implementação da lista de amigos, reutilizou-se a classe *Profile* oferecida pelo framework.

4.2.1.6.
Registro dos componentes

O registro dos pontos adaptáveis é feito na classe principal do sistema, *BuddySpaceServer,* obtendo-se uma referência à instância única de *ComponentRegistry(CR).* É necessário apenas registrar as classes *BuddySpaceServer* e *RegionManager.* Os outros pontos adaptáveis reutilizados do framework, tais como *TcpSocket, CommunicationFactory* e *ManualPositioning* já são registrados automaticamente em *CR.*

4.2.1.7.
Comandos e Fluxo de Comandos

Um usuário pode executar várias ações no sistema *BuddySpace*, como criar uma nova conta, informar sua localização corrente, conectar-se/desconectar-se do sistema, além de receber do servidor mensagens de erro e informação sobre a mudança de localização de um usuário cadastrado na sua lista de amigos. Para executar essas ações, foram reutilizados os comandos correspondentes oferecidos pelo framework. Com exceção de *LocationCommand,* que é roteado para *Positioning,* conforme descrito na Seção 4.2.1.5, os outros comandos são tratados em *BuddySpaceServer.*

Como resposta a um comando *Login*, o framework oferece o comando *LoginResponse,* que, além de informar ao cliente se o login foi bem sucedido, permite que o servidor envie uma mensagem de boas-vindas, a lista das localidades existentes e a localização corrente do usuário. Entretanto, *BuddySpace* necessita ainda que a lista de amigos do usuário seja enviada, juntamente com a localização de cada um desses amigos. Assim, um novo comando chamado *BSLoginResponse* teve que ser criado, estendendo-se a classe *LoginResponse* e criando um novo atributo para guardar essa informação extra.

Os usuários podem também enviar mensagens instantâneas para outros usuários cadastrados na sua lista de amigos. Para isso, teve que ser criada uma subclasse de *Command* chamada *InstantMessage,* cujos atributos guardam o remetente, o destinatário e o texto da mensagem. Quando um comando desse tipo chega no servidor, *BuddySpaceServer* verifica o destinatário da mensagem e a

envia para o objeto *Buddy* que o representa no sistema. Este, por sua vez, se encarrega de enviar a mensagem para seu usuário.

4.2.1.8. Execução do Servidor

O servidor da aplicação é executado, a partir da classe *BuddySpaceServer,* da mesma forma como o servidor *Nita*, descrito na Seção 4.1.1.8.

A Figura 14 mostra o diagrama de classes simplificado do servidor *BuddySpace.*

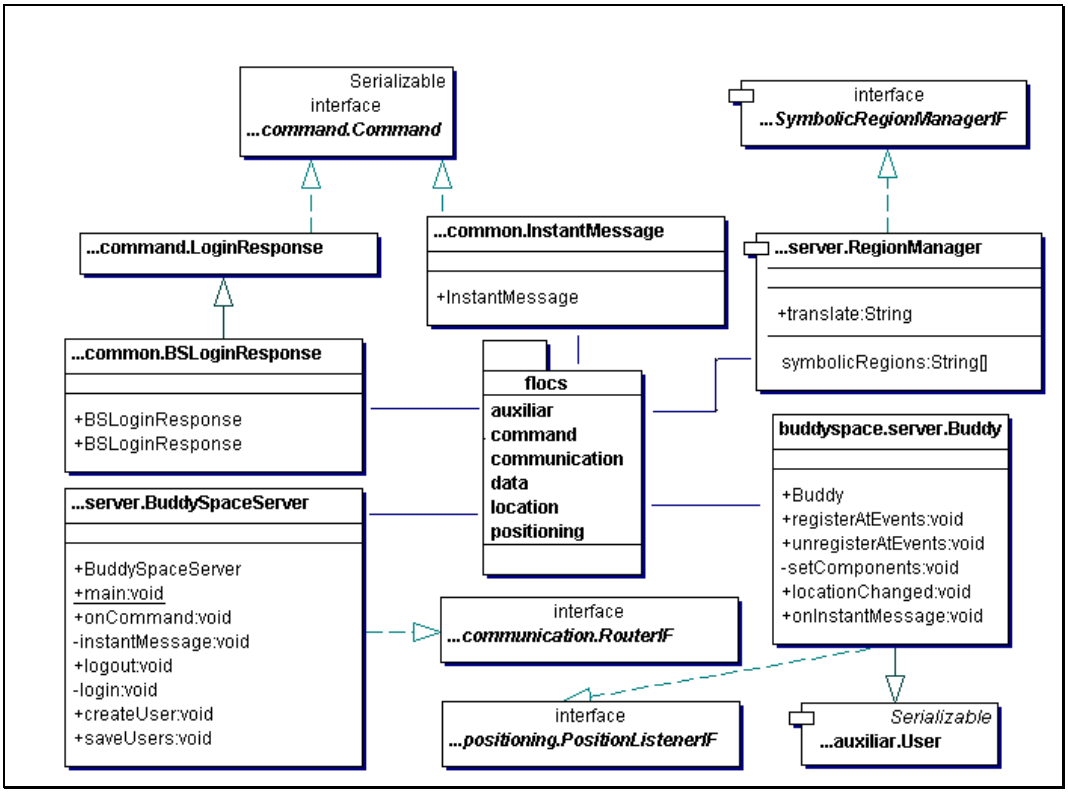

Figura 14 - Diagrama de classes simplificado do servidor *BuddySpace.*

4.2.2. O Cliente

O cliente *BuddySpace* foi desenvolvido em J2SE e pode ser utilizado em notebooks dotados de uma interface de rede sem fio.

4.2.2.1. Arquitetura

Da mesma forma que o *Nita,* no desenvolvimento do cliente *BuddySpace*, foram reutilizados os serviços de comunicação e persistência de dados e as classes de comandos oferecidos pelo framework. Foi, assim, necessário implementar apenas a interface gráfica com o usuário e a lógica de acesso a esses serviços.

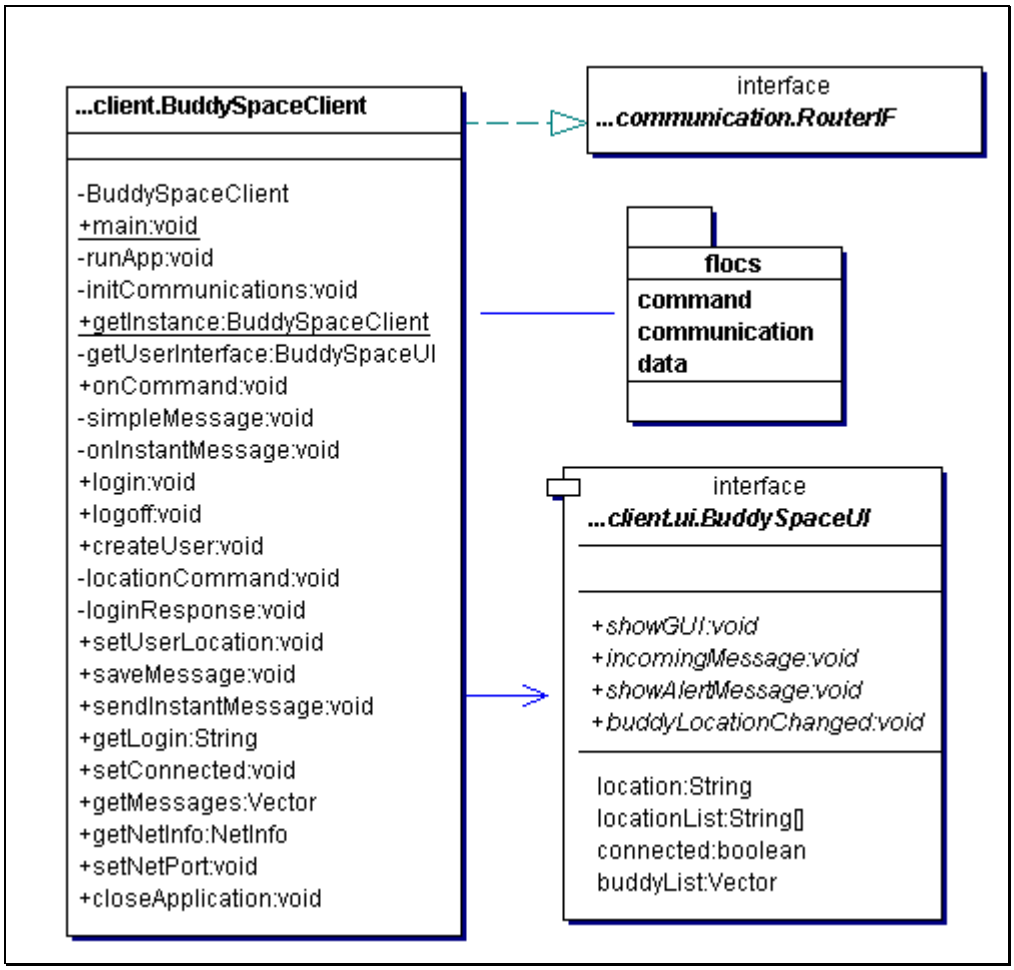

Figura 15 - Diagrama de classes simplificado do cliente *BuddySpace*.

A classe principal é *BuddySpaceClient,* que comunica-se com todos os outros componentes. Como apenas uma única instância dessa classe deve existir no sistema, ela foi criada com o padrão de projeto *Singleton*. *BuddySpaceClient* implementa a interface *RouterIF* para receber os comandos do servidor que chegam pela rede através de *TcpSocket* e utiliza *CommunicationService* para enviá-los. Além disso, persiste os dados do usuário, tais como login e senha, e as mensagens que este escolhe salvar com o uso da classe *FileService.*

BuddySpaceClient também é a responsável por controlar a interface gráfica com o usuário. Grande parte desta é composta de um mapa onde é mostrada a localização dos usuários cadastrados na lista de amigos do usuário móvel. Clicando-se na foto de um amigo, pode-se enviar uma mensagem instantânea para ele.

O mapa mostrado na interface gráfica é carregado pela classe *MapPane*, do pacote *buddyspace.client.ui.* Nesta mesma classe, as regiões simbólicas são mapeadas para pontos no mapa.

A Figura 15 mostra o diagrama de classes simplificado do cliente *BuddySpace.*

4.2.2.2. Exemplo de Cenário

Quando o cliente *BuddySpace* é executado, uma tela de login aparece para que o usuário se conecte ao sistema (Fig. 16). Quando ele, após informar seu login e senha, aperta o botão de *OK, BuddySpaceClient* obtém esses dados da interface gráfica, cria um comando *Login* e o envia para o servidor através de *CommunicationService.*

No servidor, o comando é recebido por *TcpSocket* que o repassa para *BuddySpaceClient (BSC)*, implementação de *RouterIF. BSC* verifica, então, o tipo de comando. Como essa mesma classe é a responsável pela autenticação de usuários, o comando é repassado para seu método *login.* Se a senha estiver correta, um objeto da classe *Buddy* é criado para representar o usuário no sistema. *Buddy* registra o usuário em *CommunicationService*, para que possa enviar dados via rede para o cliente móvel, e em *Positioning,* para notificá-lo da mudança de sua localização. Além disso, registra em *Positioning* o interesse pela mudança de localização dos outros usuários cadastrados em sua lista de amigos. Depois disso, um comando *BSLoginResponse* é criado contendo, além de outros parâmetros, a lista de amigos do usuário juntamente com a localização de cada um deles. Esse comando é enviado então ao cliente.

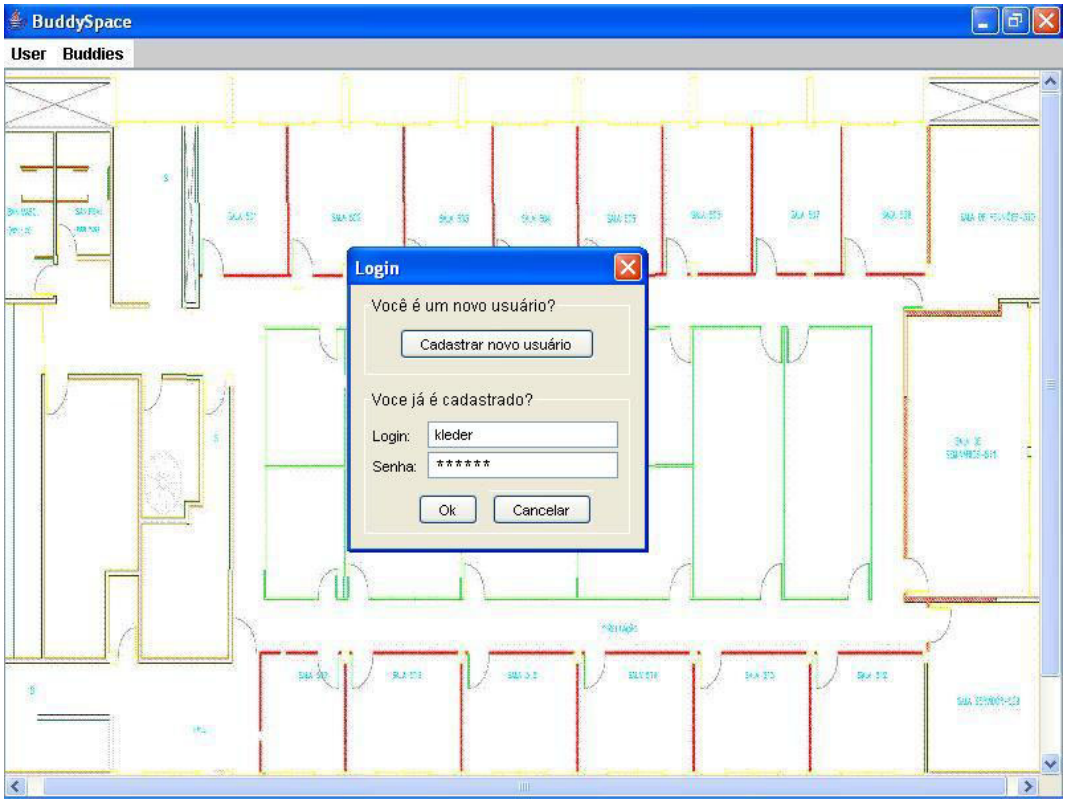

Figura 16 – Tela de login do *BuddySpace*.

No cliente, *BSLoginResponse* é recebido por *TcpSocket* que o repassa para *BuddySpaceClient.* Este verifica se o login foi um sucesso. Se não foi, mostra uma mensagem de erro. Caso contrário, exibe um mapa que mostra a localização corrente de todos os usuários cadastrados na lista de amigos recebida (Fig. 17).

O usuário pode enviar uma mensagem instantânea para um amigo dando um duplo clique na foto dele (Fig. 18). Quando o botão *OK* é clicado, *BuddySpaceClient* obtém as informações entradas na interface gráfica (texto, destinatário) e cria um comando *InstantMessage* para ser enviado ao servidor. No servidor, *BuddySpaceServer* recebe o comando de *TcpSocket*, verifica o destinatário e o envia para o objeto *Buddy* que representa o destinatário no sistema. Este se encarrega de enviar a mensagem para seu usuário (Fig. 19).

Como *BuddySpace* usa posicionamento manual, o usuário deve informar sua localização ao sistema (Fig. 20). *BuddySpaceClient* obtém a localização, cria um comando *LocationCommand* e o envia ao servidor. No servidor, *BuddySpaceServer,* roteia esse comando para *Positioning,*chamando seu método *setUserLocation.* Assim, *Positioning* notifica os interessados que esse usuário mudou de localização.

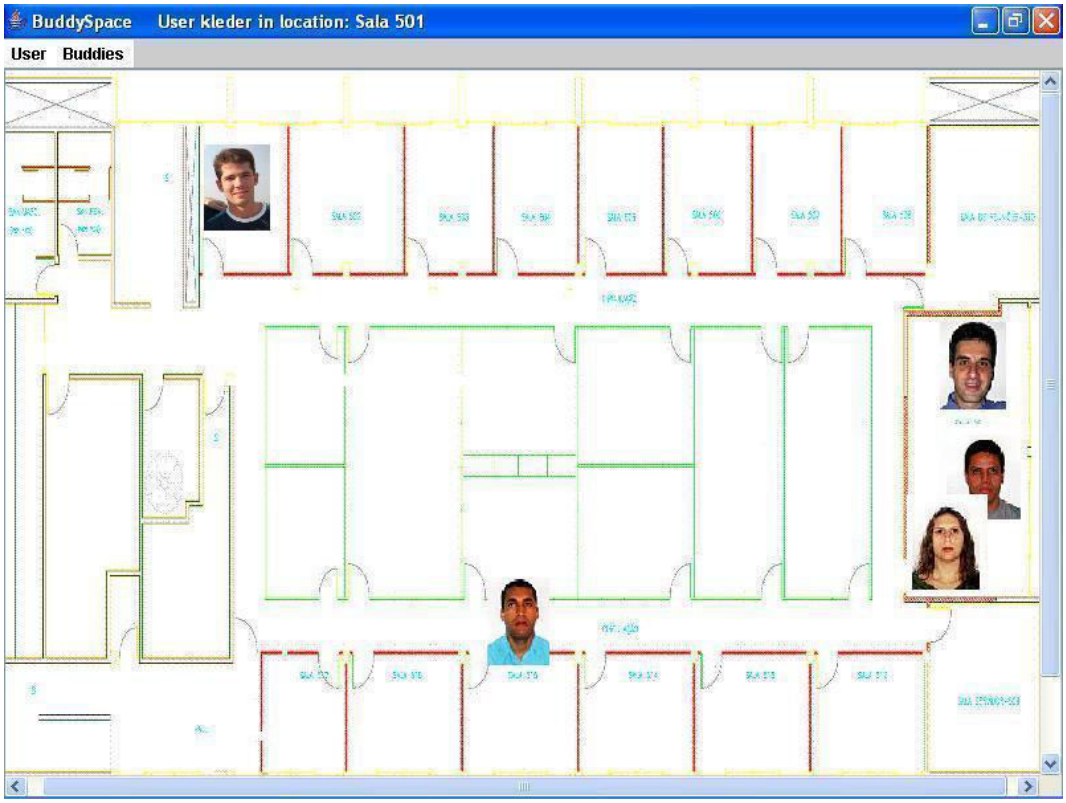

Figura 17 - Localização dos usuários cadastrados na lista de amigos.

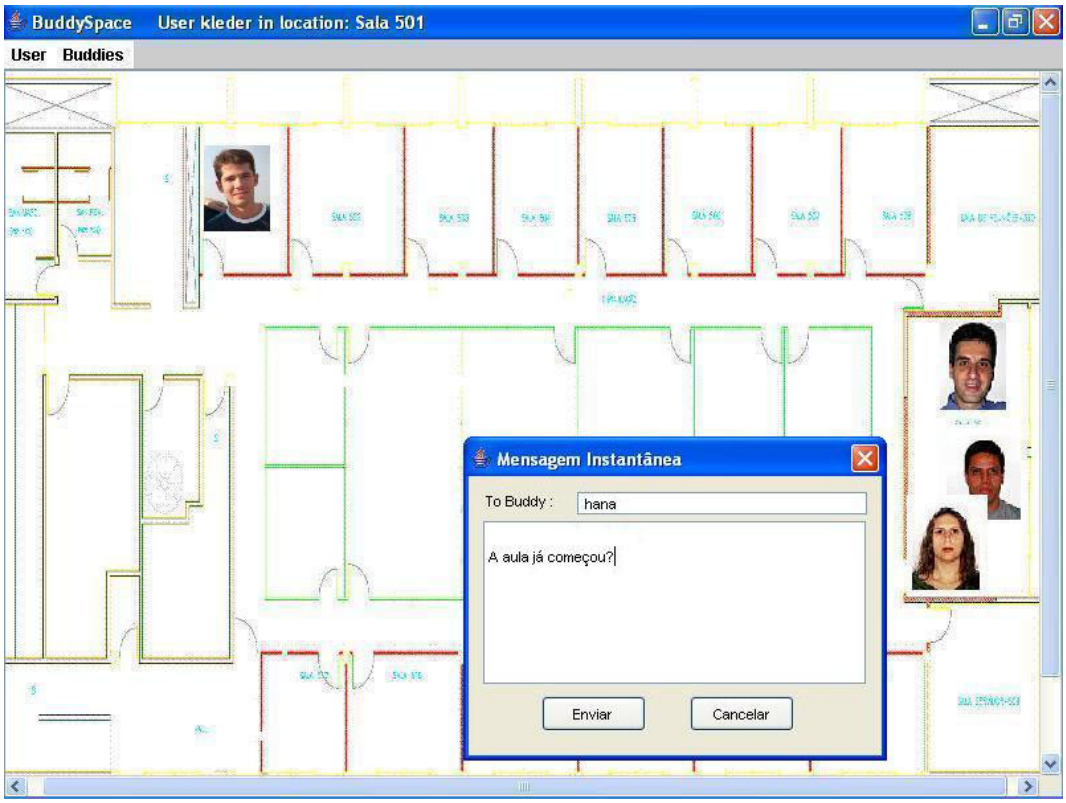

Figura 18 - Envio de mensagem instantânea.

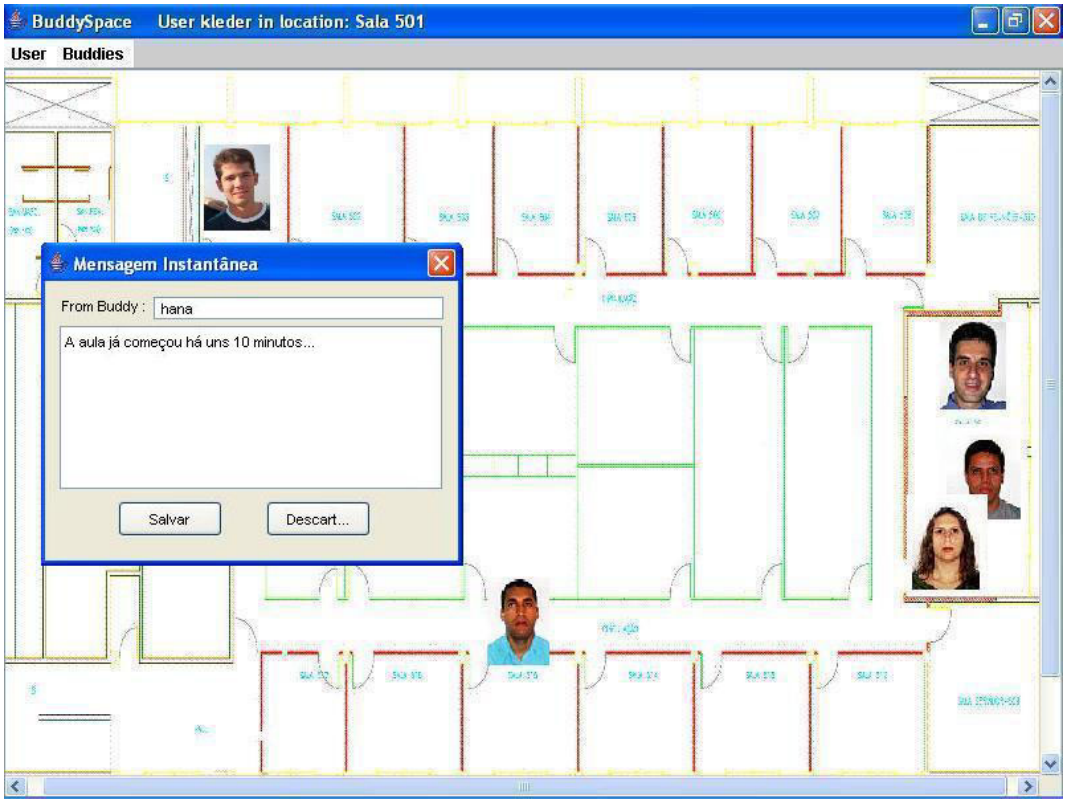

Figura 19 - Recebimento de mensagem instantânea.

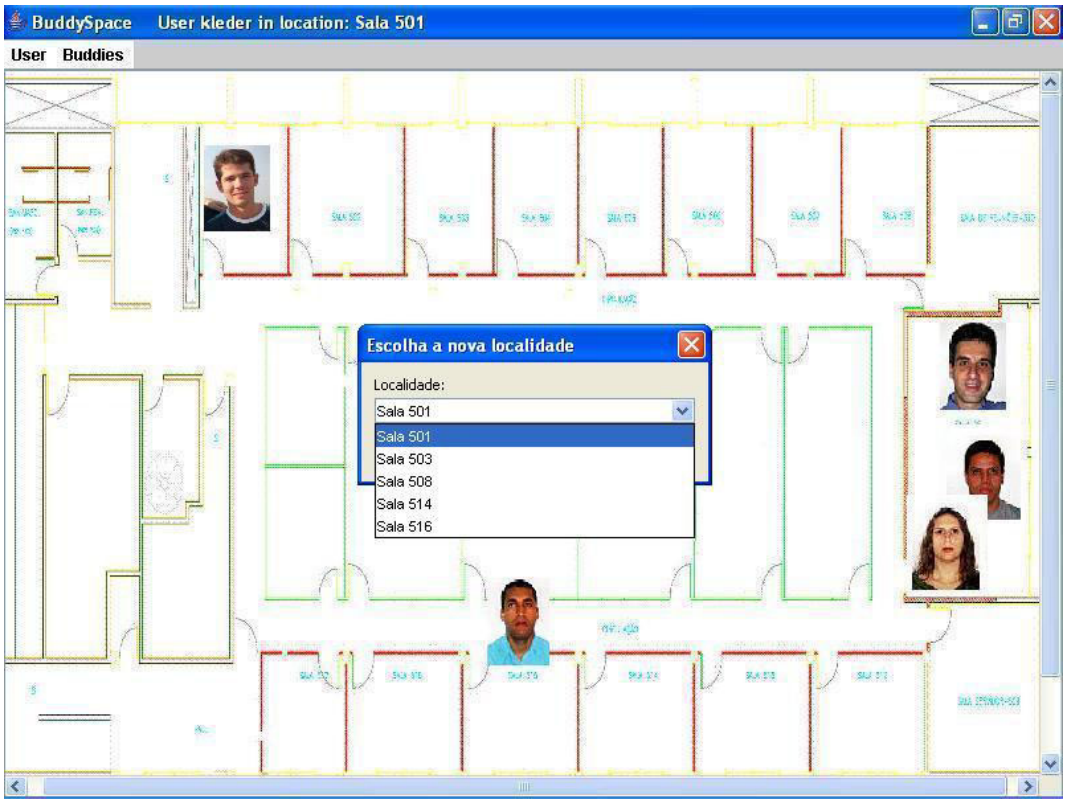

Figura 20 - Usuário informa que sua localização mudou.

4.2.3.Conclusões

A Tabela 5 mostra a reutilização de classes na instanciação do *BuddySpace*.

Pacote	Interfaces Implementadas	Classes Estendidas	Classes Reutilizadas
Communication (Servidor/Cliente)	RouterIF (BuddySpaceServer/BuddySpaceClient)		CommunicationService, CommunicationFactory, TcpSocket, NetInfo
Positioning (Servidor)	PositionListenerIF (Buddy), SymbolicRegionManagerIF (RegionManager)		Positioning, ManualPositioning,
Location (Servidor)			Location, LocationManager
Data (Servidor/Cliente)			FileIO, FileService
Auxiliar (Servidor)		User (Buddy)	Profile, ComponentRegistry
Command (Servidor/Cliente)		Command (InstantMessage) LoginResponse (BSLoginResponse)	CreateUser, LocationCommand, Login, Logout, ErrorMessage

Tabela 5 – Reutilização de classes na instanciação de *BuddySpace*.

Observando a Tabela 5, nota-se que a exemplo da aplicação *Nita*, também neste caso houve um alto grau de reutilização de código. No servidor, apenas *BuddySpaceServer* e *RegionManager* tiveram que ser efetivamente criadas. *Buddy* estendeu *User* apenas para tratar os eventos da interface *PositionListenerIF*. Além disso, duas classes de comandos tiveram que ser estendidas. Todas as demais classes foram totalmente reutilizadas.

No cliente, *BuddySpaceClient* serve apenas de classe controladora, criando comandos a partir dos dados fornecidos pelo usuário através da interface gráfica e repassando-os ao serviço de comunicação do framework para ser enviado ao servidor. Assim como no *Nita*, na instanciação do cliente *BuddySpace*, a complexidade maior está na criação da interface gráfica com o usuário.

Ao invés de *BuddySpace* usar posicionamento manual, poderia utilizar uma tecnologia de localização centrada na rede, como, por exemplo, o serviço *LIS* do *middleware MoCA,* usado pelo *Nita*. O framework permite que tal alteração seja feita facilmente, bastando para isso registrar em *ComponentRegistry* as classes *LisAdapter* e *LisRegionManager.* Estas irão substituir as classes *ManualPositioning* e *RegionManager*, respectivamente. Além disso, quando se

for registrar o usuário na tecnologia de localização, através do método *registerUser* em *Positioning,* deve-se passar como objeto de configuração o endereço da placa de rede do dispositivo móvel, informação essa requisitada pelo *LIS*.

Além da tecnologia de localização, pode-se também alterar facilmente o ponto adaptável de comunicação. Por exemplo, para utilizar a mesma forma de comunicação do *Nita,* basta registrar em *ComponentRegistry* a classe *MocaPubSub,* que irá então substituir *TcpSocket.*

A facilidade com que a tecnologia de localização e o tipo de comunicação foram alterados, além da possibilidade de reutilização de classes, demonstra que os pontos adaptáveis do framework são de fato flexíveis e genéricos.