RODRIGO CANTÚ POLO

CACHE COOPERATIVO APLICADO AO PROTOCOLO GIOP EM REDES AD HOC MÓVEIS

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

CURITIBA

2009

RODRIGO CANTÚ POLO

CACHE COOPERATIVO APLICADO AO PROTOCOLO GIOP EM REDES AD HOC MÓVEIS

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

Área de Concentração: *Sistemas Distribuídos*

Orientador: Prof. Dr. Luiz Lima Júnior

CURITIBA

2009

Polo, Rodrigo Cantú
P778c Cache cooperativo aplicado ao protocolo GIOP em redes AD HOC móveis /
2009 Rodrigo Cantú Polo ; orientador, Luiz Lima Júnior. – 2009.
xiv, 86 p. : il. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2009
Bibliografia: p. 82-86

1. Memória cache. 2. Redes de computação – Protocolos. 3. CORBA (Arquitetura de computador). 4. Sistemas de comunicação sem fio. I. Lima Júnior, Luiz Augusto de Paula. II. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. III. Título.

CDD 20. ed. – 004.62

Pontifícia Universidade Católica do Paraná
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Informática

ATA DE DEFESA DE DISSERTAÇÃO DE MESTRADO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DEFESA DE DISSERTAÇÃO Nº 07/2009

Aos 02 dias do mês de março de 2009 realizou-se a sessão pública de Defesa da Dissertação **"Cache Cooperativo Aplicado ao Protocolo GIOP em Redes AD HOC Móveis"**, apresentada pelo aluno **Rodrigo Cantú Polo** como requisito parcial para a obtenção do título de Mestre em Informática, perante uma Banca Examinadora composta pelos seguintes membros:

Prof. Dr. Luiz Augusto de Paula Lima Junior PUCPR (Orientador)	*(assinatura)*	aprovado *(aprov/reprov.)*
Prof. Dr. Alcides Calsavara PUCPR	Alcides Calsavara	APROVADO
Prof. Dr. Islene C. Garcia UNICAMP	Islene Garcia	aprovado

Conforme as normas regimentais do PPGIa e da PUCPR, o trabalho apresentado foi considerado aprovado (*aprovado/reprovado*), segundo avaliação da maioria dos membros desta Banca Examinadora. Este resultado está condicionado ao cumprimento integral das solicitações da Banca Examinadora registradas no Livro de Defesas do programa.

Prof. Dr. Mauro Sérgio Pereira Fonseca
Diretor do Programa de Pós-Graduação em Informática

Programa de Pós-Graduação em Informática PPGIa PUCPR

Rua Imaculada Conceição, 1155 Prado Velho CEP 80215 901 Curitiba Paraná Brasil
Tel.: (41) 3271 1669 Fax: (41) 3271 2121 www.ppgia.pucpr.br

Dedicatória

À minha esposa, Iris, pela compreensão, ajuda, amor e carinho.

À minha família, principalmente aos meus pais e à minha avó, por tudo o que sou.

A todos os que se esforçam para a realização dos seus sonhos.

Agradecimentos

À minha família: Iris, Marco, Dalva, Larissa, Paulo, Dora, Beatriz e Carlos, por toda a compreensão, suporte e companheirismo. Aos meus sobrinhos, Guilherme e Caio, pelas alegrias. Ao meu orientador, Prof. Dr. Luiz Lima Júnior pelos incentivos, esforço, idéias e, principalmente, pela paciência. Aos meus amigos e colegas da área de P&D da Nokia Siemens Networks de Curitiba, da Genband de Curitiba e do PPGIA pelas idéias, companheirismo e incentivo. À área de P&D da atual Nokia Siemens Networks de Curitiba, antiga área de P&D da antiga Siemens Carrier de Curitiba, pela oportunidade, viabilização e suporte financeiro aos meus estudos neste programa de Mestrado, sem os quais o cumprimento deste programa não seria possível. À Siemens Enterprise Networks de Curitiba, onde trabalho atualmente, pelo tempo imprescindível dado a mim para a conclusão desta dissertação. E a todos os que contribuíram, de forma direta ou indireta, para a realização deste trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

AODV	*Ad hoc On-Demand Distance Vector Routing*
BFS	*Breadth First Search*
CDR	*Common Data Representation*
CORBA	*Common Object Request Broker Architecture*
DNS	*Domain Name Server*
DSR	*Dynamic Source Routing*
EJB	*Enterprise Java Beans*
GCM	*Global-cache-miss initiated Caching Management*
GIOP	*General Inter ORB Protocol*
ICP	*Internet Cache Protocol*
IDL	*Interface Definition Language*
IEEE	*Institute of Electrical and Electronics Engineers*
IETF	*Internet Engineering Task Force*
IIOP	*Internet Inter-ORB Protocol*
IOR	*Interoperable Object Reference*
IP	*Internet Protocol*
KR	*Keep Requesting*
MANET	*Mobile Ad Hoc Network*
MCM	*Motion-aware Caching Management*
MobEYE	*Mobile intercEpting proxY cachE*
ORB	*Object Request Broker*
OSI	*Open Systems Interconnection*
P2P	*Peer To Peer*
PAN	*Personal Área Network*
PDA	*Personal Digital Assistant*
RIOR	*Routing IOR*
RMI	*Remote Method Invocation*

RMI-IIOP	*Remote Method Invocation over Internet Inter-ORB Protocol*
RPC	*Remote Procedure Call*
SOAP	*Simple Object Access Protocol*
TCP	*Transmission Control Protocol*
TTL	*Time To Live*
UDP	*User Datagram Protocol*
URL	*Uniform Resource Locator*
Wi-Fi	*Wireless Fidelity*
ZIOP	*GIOP Compression*

Resumo

Redes sem fio móveis estão cada vez mais populares no nosso cotidiano. Estas redes dinâmicas, que não necessitam de nenhuma infra-estrutura de operação ou gerenciamento centralizado, são conhecidas como redes *ad hoc* móveis (*Mobile Ad hoc Networks*, MANETS). Devido à instabilidade de rotas neste tipo de rede, desconexões entre dispositivos móveis são comuns, o que dificulta a transmissão de dados entre elementos de rede distantes entre si. A técnica de *cache* cooperativo visa melhorar este cenário, fazendo com que dispositivos intermediários à comunicação entre outros dispositivos de rede possam participar da transmissão, enviando dados previamente armazenados em *cache* ao dispositivo cliente, referentes à sua requisição. Apesar do crescente interesse em pesquisas sobre *cache* cooperativo aplicado a redes *ad hoc*, pouco se fez com relação à aplicação desta técnica na otimização de transmissões entre ORBs (*Object Request Brokers*), pertencentes à especificação CORBA (*Common Object Request Broker Architecture*), principalmente no contexto de redes móveis *ad hoc*. Este trabalho tem como objetivo aplicar as técnicas de *cache* cooperativo especificamente no protocolo GIOP (*General Inter ORB Protocol*), da especificação CORBA, de forma a otimizar a transmissão entre dispositivos que o utilizem no contexto de MANETs. Um algoritmo é definido e sua arquitetura é especificada, assim como uma comparação entre cenários de aplicação e os resultados da simulação feita. Também é apresentada a contribuição científica sobre fragmentação de *cache*, que faz parte da especificação do algoritmo.

Palavras-Chave: MANETs, *cache* cooperativo, CORBA, GIOP, redes *ad hoc*.

Abstract

Mobile wireless networks are becoming increasingly popular nowadays. These dynamic networks, which do not need any operational infrastructure or centralized management, are known as mobile ad hoc networks (MANETs). Due to route instability on this kind of networks, disconnections between mobile devices are usual, which makes the data transmissions between distant network devices more difficult. The cooperative caching technique aims to improve this scenario, where intermediate devices to a client-server node communication can join the transmission, sending previously stored cached data to the client device, related to the client's request. Despite the increasing interest in cooperative cache research applied to ad hoc networks, little has been done to apply this technique to optimize communications between ORBs (Object Request Brokers), which are part of the CORBA (Common Object Request Broker Architecture) specification, especially in the context of mobile ad hoc networks. This work aims to apply cooperative cache techniques to GIOP (General Inter ORB Protocol), from CORBA specification, in order to optimize transmission between devices which use GIOP in the context of MANETs. An algorithm is defined and its architecture is specified, as well as a comparison between application scenarios and the simulation results. The scientific contribution related to fragmented caching is also presented, which is part of the algorithm’s specification.

Keywords: (MANETs, cooperative cache, CORBA, GIOP, ad hoc networks).

Capítulo 1

Introdução

Redes sem fio móveis estão cada vez mais presentes no nosso cotidiano. As redes celulares atuais são o exemplo mais comum, porém observamos uma crescente diversificação deste tipo de redes. PDAs (*Personal Digital Assistants*), *notebooks* e *smart phones Wi-Fi* (*Wireless Fidelity*), são exemplos de dispositivos que se comunicam de forma completamente descentralizada. Estes aparelhos podem utilizar os próprios dispositivos móveis de sua vizinhança para que possam se comunicar com outros aparelhos que se encontrem a uma distância fora do alcance de sua interface de rede sem fio. Redes assim formadas, que não necessitam de nenhuma infra-estrutura de operação ou gerenciamento centralizado, são conhecidas como redes *ad hoc* móveis (*Mobile Ad hoc Networks*, MANETS) [FRO00]. O termo *ad hoc* se refere a redes dinâmicas, sem uma unidade centralizadora, com seus nós se conectando e se desconectando freqüentemente. Cada nó atua também como um roteador, e é responsável pelo encaminhamento de mensagens de dispositivos vizinhos para outros destinos.

Neste contexto dinâmico e imprevisível, nós que estejam obtendo dados de um nó fonte podem ter a sua conexão com este nó interrompida por causa da desconexão de algum nó intermediário, que fazia parte da rota de entrega das mensagens. A sobrecarga do nó fonte também pode fazer com que a obtenção de dados a partir do mesmo se torne mais lenta. Para a resolução de problemas como estes, nós intermediários podem armazenar a resposta de nós fonte em sua memória para que, caso o nó fonte não possa responder ao pedido do nó cliente, estes nós intermediários possam colaborar entre si de forma a responder à requisição feita pelo cliente. Esta técnica, conhecida como *cache* cooperativo, tem se mostrado bastante promissora, como pode ser visto em [YIN04] e [WU06].

Por outro lado, a arquitetura CORBA (*Common Object Request Broker Architecture*) tem sido tradicionalmente utilizada em redes estruturadas para prover interoperabilidade entre objetos distribuídos, possivelmente localizados em ambientes heterogêneos. Como os dispositivos móveis, que fazem parte do contexto de MANETs, possuem muito freqüentemente, hardware e sistemas operacionais distintos, a questão da interoperabilidade torna-se extremamente relevante. No entanto, só mais recentemente, com a proliferação de dispositivos móveis (fruto da miniaturização de componentes, aumento de vida das baterias e queda de preços) especificações e pesquisas têm surgido, adaptando o modelo CORBA para o ambiente móvel [OMG03][JAY04][LIM07]. Apesar de não ter mais o destaque anterior ao surgimento de outras arquiteturas de objetos distribuídos, como *Web Services*, CORBA é utilizada em várias áreas de atuação com vários casos de sucesso, como pode ser visto na Seção 2.10. E apesar do crescente estudo de sua aplicação no contexto de mobilidade, foram encontrados poucos trabalhos que descrevam a aplicação desta técnica na otimização da comunicação entre *middlewares* CORBA, principalmente no contexto de redes sem fio.

Desta forma, propõe-se um algoritmo de *cache* cooperativo para aplicações baseadas em GIOP (*General Inter-ORB Protocol*), que é o protocolo utilizado para a comunicação entre ORBs, no contexto de MANETs. Vale ressaltar que o algoritmo proposto deve ser aplicado a sistemas com informações que sofrem poucas mudanças ao longo do tempo, e que este protocolo utiliza particularidades do próprio protocolo GIOP. Além disso, uma comparação entre diferentes ambientes de aplicação do algoritmo é feita, apresentando-se também a contribuição científica sobre fragmentação de *cache* (Seção 4.2.6) e os resultados de simulação da execução do algoritmo.

1.1. Motivação

Várias técnicas de *cache* cooperativo para MANETs foram estudadas e formuladas. Porém, pouco foi feito com relação à aplicação desta técnica na arquitetura CORBA, principalmente com relação ao protocolo GIOP em redes *ad hoc* móveis (MANETs). Além disso, pouco foi encontrado a respeito da aplicação de CORBA em MANETs [LIM07]. Algumas questões de implementação nos artigos que falam especificamente sobre *cache* cooperativo também não são descritas claramente, principalmente pelos estudos correntes desta técnica serem muito teóricos, desconsiderando questões importantes, como a integração de camadas de rede e de aplicação do modelo OSI (*Open Systems Interconnection*).

Logo, a principal motivação desta pesquisa foi em suprir estes pontos em aberto. A pesquisa se faz necessária pela otimização de transmissões realizadas entre ORBs em MANETs, que é um ponto chave para a possibilidade de implantação desta tecnologia em redes sem fio dinâmicas. Embora o modelo de *cache* cooperativo desenvolvido tenha sido aplicado especificamente ao protocolo GIOP, ele pode ser aplicado também a qualquer protocolo de comunicação que possua as características descritas no capítulo 4, Seção 4.3.

1.2. Objetivos

Os objetivos principais deste trabalho são:

- Apresentar um algoritmo distribuído de *cache* cooperativo aplicado ao protocolo GIOP para a otimização de comunicações entre ORBs em MANETs;
- Apresentar a contribuição científica de fragmentação de *cache* como parte do funcionamento do algoritmo;
- Comparar diferentes cenários de implementação deste algoritmo; e
- Apresentar os resultados de simulação do algoritmo proposto.

A aplicação do algoritmo visa objetos CORBA que possuam mudanças internas de estado durante um intervalo de tempo que seja menor ou igual ao TTL (*Time To Live*) estipulado para os dados armazenados em *cache* pelo algoritmo. Caso contrário, as cópias em *cache* da resposta às invocações dos métodos destes objetos estarão sempre obsoletas, inviabilizando o controle de *cache* cooperativo. Maiores detalhes podem ser vistos na Seção 4.1.

O esclarecimento de alguns pontos importantes sobre a implementação do algoritmo em ambientes reais também faz parte dos objetivos deste trabalho.

1.3. Organização

O trabalho está organizado da seguinte forma. O Capítulo 2 descreve tecnologias fundamentais, que são referenciadas no decorrer deste trabalho, enquanto que o Capítulo 3 apresenta os trabalhos relacionados. No Capítulo 4 é apresentada a arquitetura do algoritmo de *cache* cooperativo. No Capítulo 5, é apresentado um modelo de implementação real do algoritmo, seguido dos dados de desempenho da sua simulação, assim como uma discussão sobre os resultados da simulação. Por fim, é feita uma conclusão do trabalho e uma descrição dos trabalhos futuros identificados, seguida das referências bibliográficas.

Capítulo 2

Fundamentos

Este capítulo trata de conceitos e tecnologias que fundamentaram o trabalho desta dissertação. São apresentadas descrições sobre localidade de referência (2.1), que é um conceito básico para que se possa entender o conceito e propósito de memórias *cache* (2.2). A questão de consistência de *cache* também é apresentada (2.3). Depois, são descritas as redes *ad hoc* (2.4), assim como a sua classificação, para que se possa entender o contexto onde redes *ad hoc* móveis (MANETs) (2.5) se encontram. Os pontos abertos em pesquisa sobre MANETs também são apresentados (2.6), além dos tipos de protocolos de roteamento utilizados por estas redes (2.7).

Em seguida, é descrito o conceito de *cache* cooperativo em MANETs (2.8), seguido da apresentação do modelo de interoperabilidade da arquitetura CORBA (2.9), assim como a sua aplicação em MANETs (2.9.2). Uma descrição sobre interceptadores portáveis em CORBA (*portable interceptors*) também é feita (Seção 2.9.3), além de uma discussão sobre vantagens e desvantagens de CORBA (Seção 2.10) e de uma comparação entre a arquitetura *Ad Hoc* CORBA e o modelo CORBA convencional, com relação a sua adaptação em MANETs (Seção 2.11). Por fim, é descrito o conceito de interceptação de *cache* ou de *proxy* (2.12), seguido do resumo do capítulo (2.13).

2.1. Localidade de referência

Para que se possa compreender o conceito de *cache*, é importante falar sobre o conceito de localidade de referência [DEN05]. Este princípio consiste no processo de acessar um único recurso múltiplas vezes. Existem três tipos básicos de referência de localidade, que são: temporal, espacial e seqüencial. Na localidade temporal, um recurso que é acessado em

certo instante terá alta probabilidade de ser acessado novamente no futuro. Na localidade espacial, é mais provável que certo recurso seja acessado se um recurso próximo a ele já foi acessado anteriormente. Por fim, na localidade seqüencial a memória é acessada seqüencialmente.

Em computação, a memória *cache* é um simples exemplo de utilização do princípio de localidade temporal. Este tipo de memória contém pouca capacidade, porém possui um acesso rápido. É utilizada para armazenar dados referenciados recentemente, o que leva a grandes ganhos de desempenho. Segue uma descrição sobre este tipo de memória na próxima seção.

2.2. Memória *cache*

Uma memória *cache* [GEN04] é uma coleção de dados que foram computados ou obtidos anteriormente, onde sua obtenção pelos mesmos meios anteriores pode gerar consideráveis problemas de desempenho no sistema, comparando-se a apenas ler os dados do *cache*. Seu desempenho é extremamente efetivo por se basear no princípio de localidade temporal.

A memória *cache* contém dados com alta probabilidade de serem utilizados novamente. Ela é constituída de múltiplas entradas, cada uma possuindo um dado, que é uma cópia da memória original. Cada uma destas cópias tem também um identificador, que especifica aonde se encontra o dado original. Um exemplo citado em [DEN05] pode ser um navegador *web*, que pode verificar seu *cache* local e procurar o conteúdo de uma certa página, onde o identificador é a URL. Neste caso, o dado é o conteúdo.

Quando a aplicação tenta acessar os dados desejados, ela inicialmente procura pelos dados na memória *cache* [GEN04]. Se uma entrada referente ao dado procurado for encontrada na memória *cache*, então o dado em questão é obtido da mesma. Esta situação é conhecida como um acerto de *cache* (*cache hit*). Caso nenhuma entrada referente ao dado seja encontrada, diz-se que ocorreu um erro de *cache* (*cache miss*). O dado então é obtido de sua localidade principal e uma nova entrada referente a ele é armazenada na memória *cache*, para que esteja disponível para o próximo acesso.

A figura 2.1 ilustra o seu funcionamento. O campo `Etiqueta` da memória *cache* contém o valor do campo `Índice` da memória principal, enquanto que o campo `Dado` da memória *cache* contém a cópia do dado contido no campo `Dado` da memória principal. Caso o dado de índice número dois seja procurado, a memória *cache* retorna o dado desejado, que

corresponde ao seu dado armazenado de índice número zero, o que leva a um caso de acerto de *cache*. Caso o dado de índice número três seja procurado, uma situação de erro de *cache* acontecerá, pois este dado não está presente na memória *cache*. Este dado será então armazenado na memória *cache*, para que ela possa servir futuras consultas a ele.

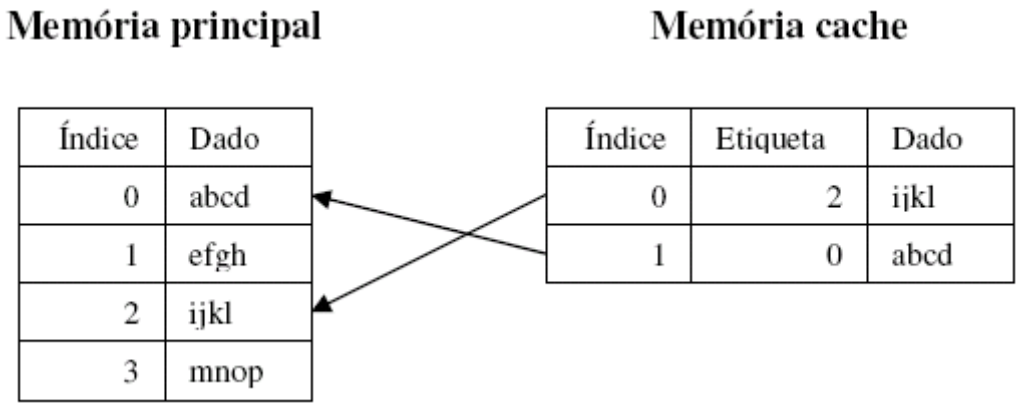

Figura 2.1: Diagrama de uma memória *cache* de CPU

Se a memória *cache* tiver capacidade limitada, parte de suas entradas terá de ser descartada. Vários critérios podem ser adotados para essa remoção, que podem ser por freqüência de acessos, tamanho dos blocos armazenados, etc.

Caso o dado original seja modificado, existem métodos de coerência que atualizam as memórias *cache* de acordo com estas modificações, evitando que fiquem desatualizadas [GEN04]. Este conceito de coerência é usualmente aplicado ao tratamento de *cache* em microprocessadores que, nesta proposta, é chamado de consistência de *cache*, devido à referência constante a esse termo na literatura sobre *cache* cooperativo.

2.3. Consistência de *cache*

Um método de consistência de *cache* deve prover atualizações nas cópias do dado original depois que o mesmo for alterado [YIN04]. Para evitar o grande número de troca de mensagens quando os dados originais são alterados, um limite no desvio entre o dado do nó fonte e o dado em *cache* pode ser considerado. Exemplos dessas tolerâncias podem ser citados, como o *cache* de uma página *web* que não pode ficar defasado por mais de uma hora, ou o preço da cotação de uma ação, que não pode ser impreciso por mais de uma hora [COO06].

Quando *cache* é utilizado, problemas de consistência de *cache* devem ser resolvidos para assegurar que clientes apenas consigam ter acesso a estados válidos dos dados, ou ao menos não acessar dados obsoletos sem conhecimento disso, de acordo com as regras do modelo de consistência. Problemas relacionados à consistência de *cache* foram estudados em vários outros sistemas, como arquiteturas de multiprocessadores, sistemas de arquivos distribuídos, memória compartilhada distribuída e sistemas de banco de dados em arquitetura cliente-servidor.

Dois modelos [YIN04] amplamente utilizados de consistência de *cache* são o modelo de consistência fraca e o modelo de consistência forte. No modelo de consistência fraca, um dado obsoleto pode ser retornado para o cliente. No modelo de consistência forte, depois que alterações são feitas nos dados originais, cópias obsoletas dos dados modificados não serão retornados para o cliente. A definição exata da finalização de uma escrita varia de acordo com a abordagem de consistência. O mecanismo de consistência fraca normalmente utilizado é o baseado em TTL, onde um cliente considera uma cópia em *cache* atualizada se o TTL não expirou. Para o modelo de consistência forte de *cache*, abordagens baseadas em invalidação e votação são utilizadas. Na abordagem de invalidação, o servidor mantém um controle dos clientes que têm o item de dados em *cache*, e envia mensagens de invalidação para os clientes quando o dado é alterado. No modo de votação, cada vez que um usuário requer um item de dados e existe uma cópia em *cache*, o respectivo nó de *cache* primeiramente contata o servidor para validar a cópia em *cache*, e depois retorna a cópia para o usuário. Como a abordagem por votação pode gerar um significante tráfego de rede, a aproximação baseada em TTL é amplamente utilizada para o modelo de consistência fraca, enquanto que o modelo baseado em invalidação é utilizado no modelo de consistência forte.

2.4. Redes *ad hoc* e sua classificação

De acordo com [FRO00], o conceito mais comum com relação a uma rede *ad hoc* é sobre uma rede formada sem nenhuma administração central, que consiste em nós móveis que utilizam interfaces de rede sem fio para enviar pacotes de dados. Como os nós nestas redes podem atuar tanto como nós normais quanto como roteadores, estes nós podem encaminhar pacotes em nome de outros nós e suportar aplicações de usuários, simultaneamente.

Em redes celulares dos dias atuais, que dependem fortemente de sua infra-estrutura, a cobertura é feita por suas estações base, onde os aparelhos são gerenciados a partir de um

ponto central, com serviços integrados ao sistema. Este modelo é responsável pelo sistema de comportamento previsível e controlado das redes celulares atuais, que também compreende o modelo IEEE 802.11 baseado em infra-estrutura. Adicionalmente, são modelos baseados em apenas um único salto de distância entre a estação base e os nós da rede, o que faz com que o controle seja mais efetivo.

Quanto mais descentralizado se torna este gerenciamento, mais próximo ao conceito de redes *ad hoc* puras o modelo se aproxima. Esta tendência pode ser vista na Figura 2.2.

Ainda com relação a redes celulares, a dependência de uma administração centralizada é reduzida quando os aparelhos se comunicam com a estação base por meio de outros aparelhos, caracterizando um modo de comunicação de saltos múltiplos, denominado de redes celulares de saltos múltiplos (*cellular multihop*).

Redes completamente descentralizadas, como redes *ad hoc* móveis (MANETs), redes IEEE 802.11 operando em modo *ad hoc* e áreas de rede pessoal (*Personal Area Networks*, PANs), têm uma melhor afinidade com o modelo puro de redes *ad hoc*. A iniciativa do conceito de MANETs foi feita pela *Internet Engineering Task Force* (IETF), que também quer incluir a este modelo serviços disponíveis pela rede de infra-estrutura fixa, conectada à *internet*.

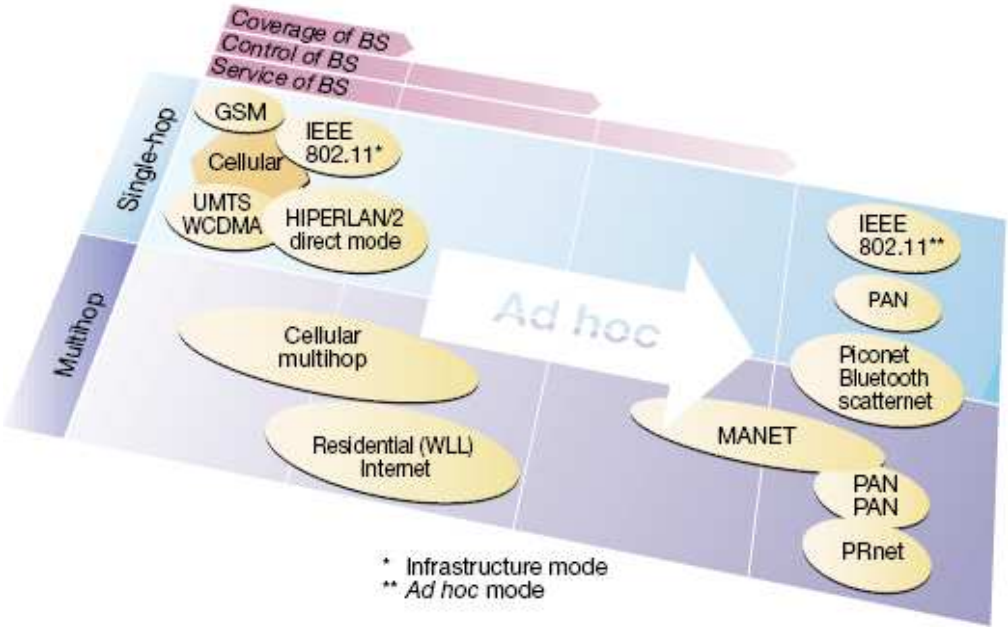

Figura 2.2: Classificação de redes *ad hoc*, de acordo com o suporte feito pela estação base e de acordo com o número de saltos possíveis para estabelecimento de conexões [FRO00].

2.5. Redes *Ad hoc* móveis (MANETs)

Segundo [FEE99], uma rede *ad hoc* móvel (MANET, Mobile *Ad hoc* Network), é um grupo de nós móveis sem fio que, cooperativamente e de forma espontânea, formam uma rede independente de qualquer infra-estrutura fixa, ou administração central. Uma MANET não tem estações base. Um nó pode se comunicar diretamente com outros nós dentro de sua área de alcance por um único salto, além de poder se comunicar indiretamente com outros nós, que estejam fora de seu alcance, por meio de nós intermediários, caracterizando uma comunicação por saltos múltiplos. Este ambiente é caracterizado por nós com limitação de consumo de energia e de taxa de transmissão, com conexões sem fio de capacidade variável e de topologia dinâmica, o que leva a mudanças freqüentes e imprevisíveis de conectividade.

2.6. Pontos em aberto em MANETs

Redes *ad hoc* móveis geralmente são voltadas para aplicações militares e de disponibilização de dados em situações de emergência, com curto alcance e escala bem mais reduzida, e comparada a redes de computadores convencionais [TAN05]. A maioria dos protocolos *ad hoc* se encontram na camada de rede, voltados para a determinação de rotas otimizadas a elementos presentes na rede. Mesmo com vários protocolos de roteamento disponíveis, não existe uma padronização de protocolos deste tipo. Existem requisitos de desempenho em áreas como roteamento, *cache*, busca e consistência, que requerem técnicas de otimização entre camadas que utilizem "conhecimento da aplicação" para propósitos de otimização em camadas inferiores. Segurança e tolerância a faltas também são questões importantes a serem consideradas [COO06].

2.6.1. Requisitos

Segundo [BOS05], MANETs requerem protocolos otimizados para transferência e procura de dados. Para muitas aplicações, é necessário que o desempenho seja equivalente à de redes cabeadas. Estes protocolos devem também aproveitar as informações de contexto de uma tarefa a ser executada. Isto inclui conhecimento de localização, vizinhos, mobilidade e disponibilidade de energia para os nós, além da qualidade da conexão, como banda e estabilidade. Estes protocolos têm também que tratar situações que são exceções em redes cabeadas mas comuns em MANETs, como conexões perdidas, redes particionadas e nós inalcançáveis.

Logo, a MANET deve prover uma alta conectividade, onde certo nó pode acessar qualquer outro nó da rede a qualquer momento, com alta probabilidade [GER05]. De preferência, os nós das MANETs devem ter pouca mobilidade, para que o número de atualizações em suas tabelas de roteamento e outras estruturas seja reduzido.

2.6.2. Desafios

Com relação ao roteamento, a camada de redes de MANETs geralmente é inadequada para prover os serviços desejados pelas aplicações [GER05]. A imprevisibilidade dos canais de rádio, combinada com a mobilidade dos usuários, traz sérios desafios ao roteamento, requerendo intervenção das camadas superiores.

Do ponto de vista da aplicação, o desempenho está longe do que é alcançado em redes fixas [BOS05]. As informações de contexto relacionadas ao nó em questão devem ser consideradas, o que sugere fortemente o uso da aproximação de camadas de aplicação e de rede. Informações como mobilidade podem determinar melhores rotas, e informações sobre seus vizinhos podem servir para selecionar o nó mais eficiente para armazenamento de *cache*.

2.7. Protocolos de roteamento em MANETs

Protocolos de roteamento em MANETs [FEE99] podem ser classificados de acordo com vários critérios, que são: por modelo de comunicação, que é a forma de comunicação sem fio; por estrutura, que é a forma de tratamento dos nós, onde alguns são tratados de forma diferente; por estado de informação, que caracteriza como informações de topologia de rede são encontradas em cada nó; e por sincronização, que caracteriza, em cada nó, como são mantidas as informações de roteamento para cada nó destino.

O critério mais comum é por sincronização, cuja classificação é descrita a seguir.

2.7.1. Pró-ativos ou orientados a tabela

Protocolos pró-ativos, ou orientados a tabela, mantêm informações de roteamento para todos os destinos conhecidos, em cada nó fonte. Neste modelo, os nós trocam informações de roteamento periodicamente, ou por resposta a mudanças de topologia de rede. Esta técnica tem a vantagem de diminuir o tempo de espera na obtenção de uma rota quando o tráfego para um certo destino é iniciado, além de poder determinar rapidamente se um destino pode ser alcançado. Por outro lado, esta técnica consome recursos de rede significativos, onde os

recursos utilizados para estabelecer e re-estabelecer rotas não utilizadas são amplamente desperdiçados.

2.7.2. Reativos ou sob demanda

O custo em manter informações sobre rotas em um ambiente de recursos escassos e altamente dinâmico é um problema mais sério do que em redes de estrutura fixa. Como resultado, protocolos de roteamento reativo que descobrem rotas sob demanda foram propostos. Estes protocolos se baseiam na idéia de um processo de descoberta de rotas e de um processo de manutenção de rotas. O processo de descobrimento é iniciado quando uma fonte precisa de uma rota para um destino, para o qual ela envia o pedido por *broadcast*. Cada nó intermediário que recebe o pedido guarda o enlace por onde recebeu o pacote, reenviando o mesmo por *broadcast*, ignorando mensagens de *broadcast* duplicadas. Quando o pedido chega ao seu destino, a resposta é enviada para a fonte, através de informações de roteamento obtidos dos nós intermediários. Como o pedido pode chegar ao destino por outras rotas, a resposta pode ser enviada pela rota com menos saltos de distância, ou menor latência, etc.

Estes protocolos não desperdiçam recursos com manutenção de rotas desnecessárias, mas o processo de descobrimento de rotas é caro e imprevisível. Em particular, a latência decorrente deste processo é mais variável que o tempo constante de atualização de tabelas em protocolos pró-ativos.

2.7.3. Híbridos

Segundo [TAN05], estes protocolos combinam características de protocolos pró-ativos e reativos. Protocolos híbridos vêem a rede como várias zonas distintas. Nós da mesma zona adotam protocolos pró-ativos, enquanto que nós de zonas diferentes utilizam protocolos reativos para encontrarem a rota entre eles, quando necessário.

2.8. *Cache* cooperativo em MANETs

Em um ambiente centralizado, um nó fonte em uma MANET pode ficar sobrecarregado devido ao grande número de requisições [COO06]. A abordagem de *cache* cooperativo resolve este problema, onde a fonte de dados adota uma coleção de nós de *cache*. Os dados mais populares são então armazenados tanto na fonte de dados quanto nos nós de *cache*. As vantagens desta abordagem são o menor atraso, menor sobrecarga de comunicação

e a colaboração para consultas, sem sempre ter que enviar o requerimento para a fonte de dados. A Figura 2.3 ilustra um cenário de utilização de *cache* cooperativo em redes *ad hoc*, onde a rede *ad hoc* se comunica com a *Internet* através de uma estação base.

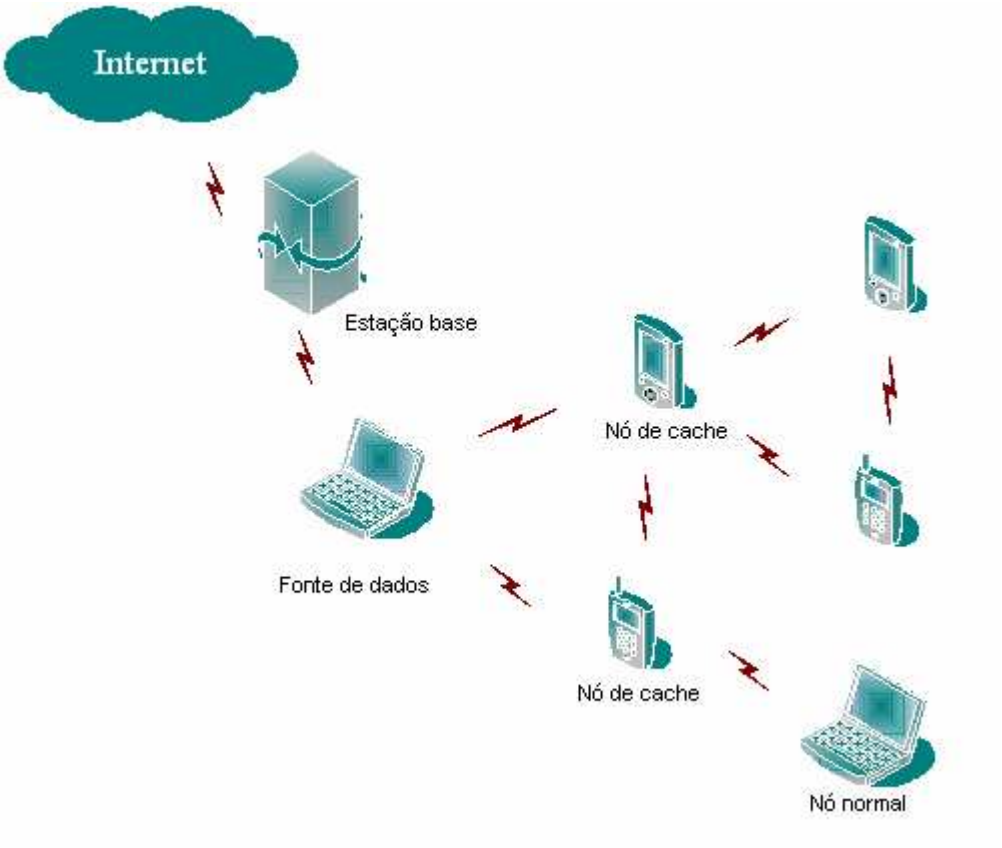

Figura 2.3: Cenário de MANET com *cache* cooperativo [COO06]

2.9. CORBA

CORBA (*Common Object Request Broker Architecture*) basicamente provê interoperabilidade entre objetos distribuídos, possivelmente localizados em ambientes heterogêneos, além de serviços distribuídos adicionais [LIM07]. Esta combinação de interoperabilidade e de serviços traz um grande suporte ao desenvolvimento de aplicações móveis distribuídas, na idéia de liberar o desenvolvedor da preocupação com detalhes de baixo nível de comunicação. Cada objeto CORBA implementa uma interface especificada pela linguagem IDL (*Interface Definition Language*). Clientes acessam objetos CORBA por referências, que podem ser obtidas por arquivos, por outros objetos ou por serviços CORBA bem conhecidos, como o serviço de nomes. Clientes enviam requerimentos para objetos CORBA e recebem respostas deles por meio do *middleware* ORB (*Object Request Broker*),

que basicamente provê a interoperabilidade entre estas entidades. A Figura 2.4 ilustra o funcionamento básico da arquitetura CORBA.

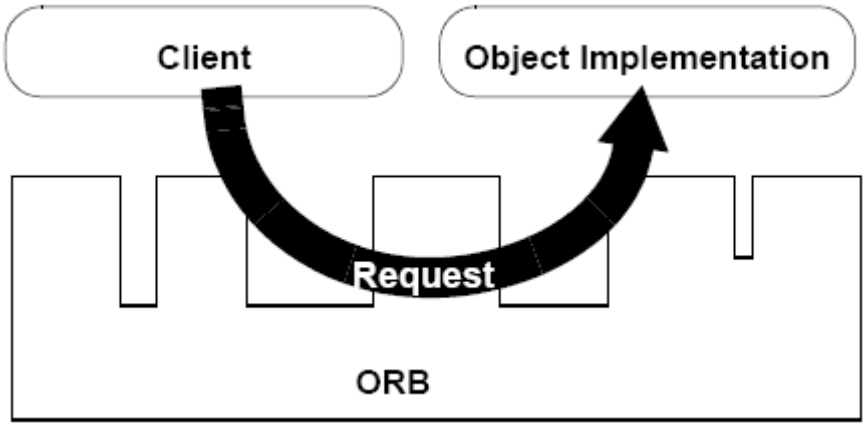

Figura 2.4: Uma requisição sendo enviada pelo ORB [OMG04]

De acordo com [OMG04], o ORB é responsável por todos os mecanismos requeridos para: localizar a implementação do objeto de acordo com um requerimento, preparar a implementação do objeto para receber o requerimento e enviar os dados ao cliente, de acordo com o requerimento. A interface que o cliente acessa é completamente independente donde o objeto está localizado, em qual linguagem de programação o mesmo foi implementado ou de qualquer outro aspecto que não esteja presente na interface do objeto.

Mensagens trocadas entre ORBs são codificadas utilizando-se o protocolo GIOP (*General Inter-ORB Protocol*) [LIM07]. O GIOP especifica oito tipos de mensagens, onde o mecanismo de RPC (*Remote Procedure Call*) é implementado por apenas dois deles, que são a mensagem de requerimento (*Request*), enviada por clientes, e a mensagem de resposta (*Reply*), enviada por servidores. Caso o conteúdo a ser transmitido por estas mensagens exceda o limite máximo definido para o protocolo GIOP, mensagens subseqüentes do tipo *Fragment* são enviadas logo em seguida às mensagens *Request* ou *Reply*.

IIOP (*Internet Inter-ORB Protocol*) é uma implementação de GIOP sobre a infra-estrutura de rede de TCP (*Transmission Control Protocol*)/IP (*Internet Protocol*), que basicamente define a codificação de IORs (*Interoperable Object Reference*). Ou seja, este protocolo define como representar informações de endereçamento TCP/IP em um IOR, de forma que o cliente possa estabelecer uma conexão com o servidor.

2.9.1. Estrutura de mensagens de resposta no protocolo GIOP

GIOP define dois tipos distintos de cadeias de *byte*s: mensagens e encapsulamentos [OMG04]. Mensagens são as unidades básicas de troca de informações no GIOP, enquanto que encapsulamentos são cadeias de *byte*s nas quais estruturas de dados IDL podem ser serializadas, independentemente de qualquer contexto particular da mensagem. A sintaxe de transferência da representação comum de dados (CDR, *Common Data Representation*) é o formato em que o GIOP representa tipos de dados de IDL.

Mensagens de resposta (*Reply*) são enviadas em resposta a mensagens de requerimento (*Request*), se é esperada alguma resposta por parte do cliente. As mensagens de resposta podem incluir valores de exceção. Nas versões de GIOP 1.0 e 1.1, o fluxo de mensagens de resposta acontece somente do servidor ao cliente.

Mensagens *Request*, *Reply* e *Fragment* são compostas por três elementos, a saber: o cabeçalho da mensagem GIOP, o cabeçalho da mensagem em questão, e o corpo da mensagem. O conteúdo do corpo da mensagem é sempre serializado no formato CDR. Nas mensagens *Reply*, o tipo dos dados do corpo da mensagem é sempre determinado pelo valor do campo *reply_status*, pertencente ao cabeçalho da mensagem. O tamanho padrão das mensagens GIOP *Fragment* é de 1024 bytes, que pode ser modificado [SUN03]. A Figura 2.5 abaixo ilustra o formato de mensagens *Request*, *Reply*, e *Fragment*.

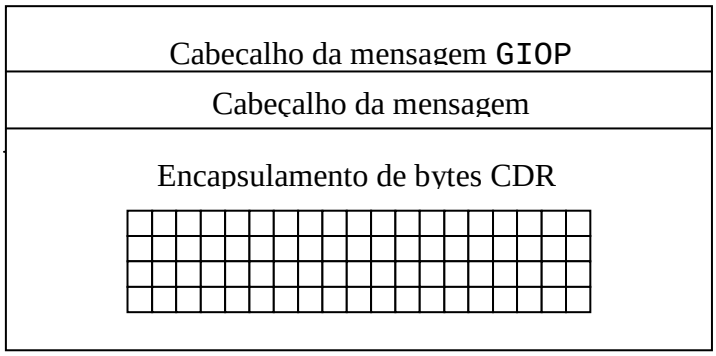

Figura 2.5: Estrutura geral das mensagens GIOP *Request, Reply* e *Fragment.*

2.9.2. CORBA em MANETs (*Ad Hoc* CORBA)

Segundo [LIM07], a mobilidade pode ser suportada em diferentes níveis (nível de conexões ou nível de IP móvel). Todavia, o nível de *middleware* é o melhor lugar para se lidar com mobilidade entre domínios de provisionamento administrativos, ou de serviços. Padrões da OMG para acesso sem fio e mobilidade de terminais em CORBA são baseados na

afirmação de que não precisam ser aplicadas modificações sobre um ORB comum, sem mobilidade, para que o mesmo possa interagir com objetos cliente e servidor, executando em um terminal móvel.

Para a adaptação do ORB a uma rede *ad hoc*, a seguinte arquitetura é proposta por [LIM07]. Primeiramente, cada terminal deve conter um servidor de nomes especializado, denominado ah-NS (*Ad Hoc Naming Service*), para que objetos possam publicar seus serviços a serem pesquisados por clientes. Se um serviço não for encontrado localmente pelo cliente, a procura é propagada a outros módulos ah-NS de outros terminais por um algoritmo BFS (*Breadth First Search*), que tem o propósito de descobrir a localização do objeto requerido e a rota a ser utilizada pelo cliente para se comunicar o terminal que mantém o objeto. A arquitetura geral do modelo é mostrada na Figura 2.6, onde os módulos ah-TB (*Ad Hoc Terminal Bridge*) são *terminal bridges* [OMG03].

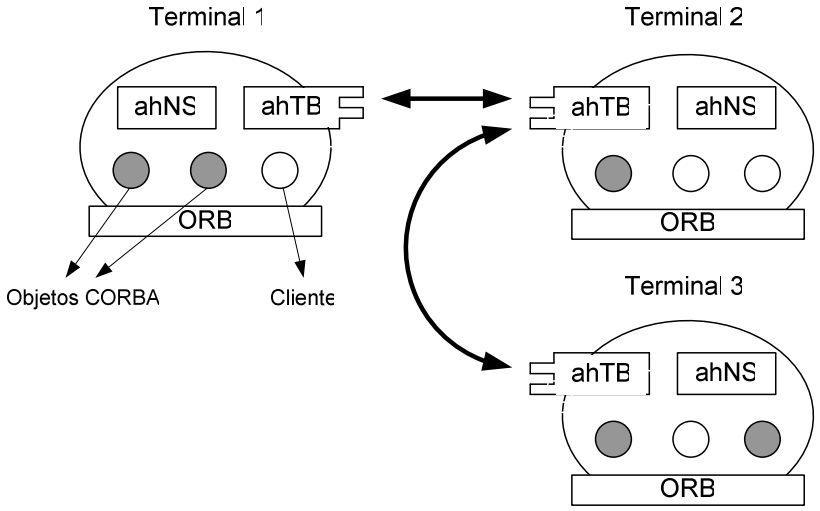

Figura 2.6: Arquitetura geral do modelo de interoperabilidade *ad hoc* utilizando CORBA [LIM07].

Os módulos de ah-NS são de extrema importância na arquitetura, pois são responsáveis por duas tarefas principais na rede *ad hoc*: publicação de serviços e pesquisa e construção de IORs de roteamento (RIORs - *Routing IORs*), que contêm informações sobre o roteamento de requisições e respostas entre dois ORBs em comunicação. A idéia do RIOR é encapsular informações de cada terminal no caminho entre dois ORBs comunicantes, para que se possa construir a rota de comunicação entre eles. Além de conter informações sobre a rota, o RIOR encapsula um IOR, necessário para a instanciação do objeto requerido.

A construção da rota é feita pelo algoritmo BFS, como ilustrado na Figura 2.7. Na sua execução, o módulo ah-NS de cada terminal que recebe a mensagem de descoberta de rotas

adiciona informações sobre o terminal na lista de perfis de roteamento (*Routing Profiles*) contida na mensagem. O propósito disto é construir a rota de conexão entre os terminais cliente e servidor, que será obtida pela cliente quando o servidor receber a mensagem e a enviar para o cliente pela rota que foi armazenada na mensagem. O algoritmo começa a busca enviando a mensagem de descoberta de rotas para os seus vizinhos, que estejam a um salto de distância. Caso o objeto não seja encontrado, a mensagem é enviada aos vizinhos dos vizinhos, e assim por diante. Um raio de atuação da busca também pode ser especificado, para evitar buscas que se estendam por toda a rede *ad hoc*.

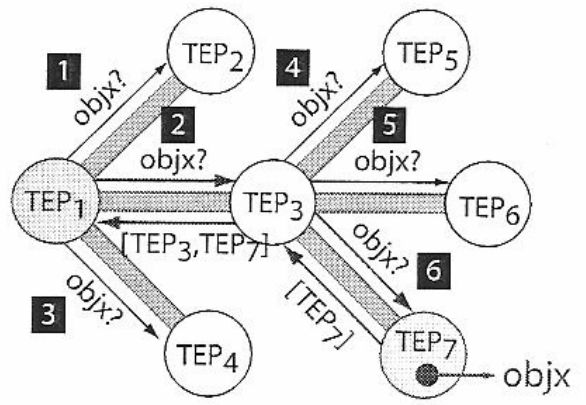

Figura 2.7: Construção do RIOR durante a procura por objx [LIM07].

Após a descoberta da rota pelo BFS, as informações sobre a RIOR construída são colocadas nas mensagens *Request* e *Reply* trocadas entre os ORBs em comunicação como itens adicionais de *service context*, que são obtidos por cada nó intermediário à comunicação entre os dois ORBs para verificação do próximo nó ao qual a mensagem deve ser encaminhada, até que a mensagem chegue ao seu destino.

2.9.3. Interceptadores Portáveis em CORBA

Implementações CORBA têm mecanismos proprietários que permitem que usuários adicionem o seu próprio código ao fluxo de execução de um ORB. Este código, conhecido como interceptador, é chamado em pontos específicos durante o processamento de requisições. O código pode inspecionar e até mesmo manipular as requisições [OMG04].

Pelo fato deste mecanismo ser extremamente flexível, a OMG padronizou os interceptadores a partir da especificação CORBA 2.4.2, sobre o nome de Interceptadores Portáveis (*Portable Interceptors*). A idéia é definir uma interface padrão para registrar e

executar códigos independentes de aplicação para, além de outras coisas, manipular *service contexts*.

Dois tipos de interceptadores são definidos: Interceptadores de Requisições e Interceptadores de IOR. Interceptadores de requisições são chamados durante o tratamento de requisições. Interceptadores de IOR são chamados quando referências de objetos são criadas, para que dados específicos de serviços possam ser adicionados a IOR recém criada.

Cinco pontos de interceptação estão no lado do cliente:

- *send_request* (envio de requisições).
- *send_poll* (envio de requisições).
- *receive_reply* (recebimento de respostas).
- *receive_exception* (recebimento de respostas).
- *receive_other* (recebimento de respostas).

Cinco pontos de interceptação estão no lado do servidor:

- *receive_request_service_contexts* (recebimento de requisições).
- *receive_request* (recebimento de requisições).
- *send_reply* (envio de respostas).
- *send_exception* (envio de respostas).
- *send_other* (envio de respostas).

A Figura 2.8 ilustra os pontos de interceptação voltados a requisições.

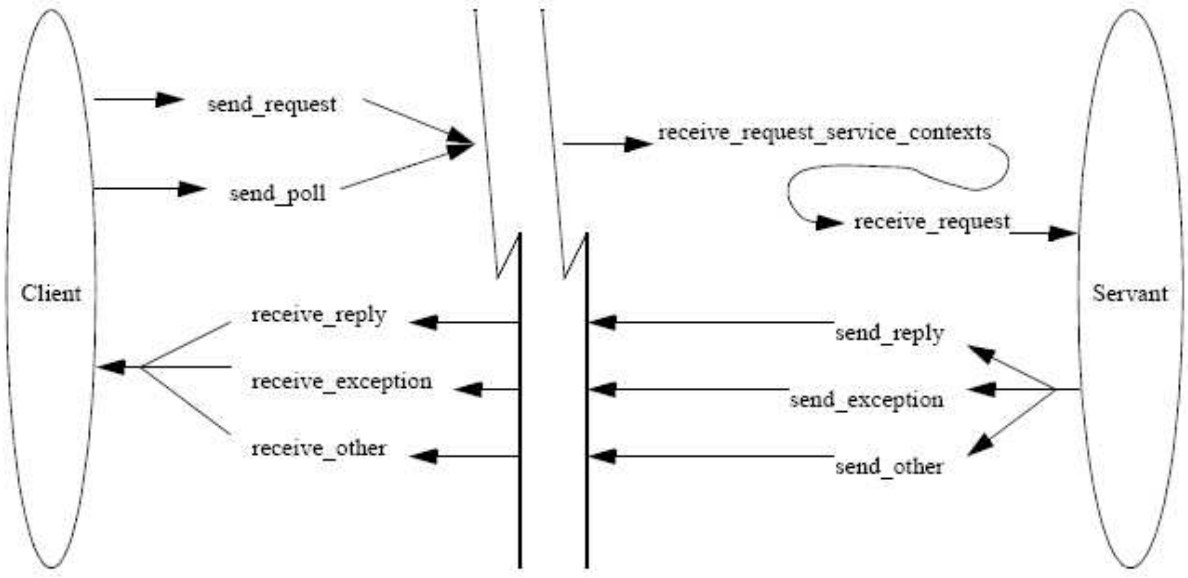

Figura 2.8: Pontos de interceptação de requisições [OMG04].

O único ponto de interceptação para interceptadores de IOR é *establish_component*. O ORB invoca este ponto de interceptação para todos os interceptadores de IOR registrados quando está montando o conjunto de componentes que está para ser incluído no *profile* IOP para uma nova referência a um objeto.

2.10. CORBA: Vantagens e desvantagens

CORBA, nos dias de hoje, não tem a mesma popularidade e destaque que tinha no final dos anos 90, com relação à utilização em larga escala em aplicativos comerciais. Isso aconteceu basicamente pelos seguintes motivos [HEN06]:

- CORBA tem uma especificação complexa. Implementar uma aplicação não trivial baseada em CORBA costuma ser difícil.
- Até a versão 3.0.3 de CORBA, de 2004, o tráfego de mensagens entre ORBs não era criptografado, fazendo com que transmissões entre aplicações fossem suscetíveis a ataques do tipo *man-in-the-middle*. Além disso, CORBA exige que algumas portas sejam liberadas no *firewall*, algo que não é necessário para *Web Services*, onde se costuma utilizar a porta 80 (HTTP), para comunicações.
- Versionamento de código para sistemas baseados em CORBA é praticamente inexistente. Quando um sistema CORBA deve ser atualizado, a atualização deve ser feita em todo o sistema, o que muitas vezes se torna inviável na prática, pois isso exige que a atualização seja feita em todos os nós da rede que utilizam o sistema CORBA.
- Ainda não existem mapeamentos de CORBA para C# ou VB.NET.

Estes fatores fizeram com que várias empresas adotassem outras plataformas, como *Web Services* e EJB (*Enterprise Java Beans*), que supriam boa parte destes pontos em aberto, mesmo não sendo necessariamente plataformas com melhores soluções técnicas.

Por outro lado, CORBA tem sido utilizada de forma crescente no desenvolvimento de sistemas embutidos e de tempo real, apesar de críticas feitas em geral ao desempenho de suas implementações. Isto porque ORBs que implementam a especificação *Real-Time* CORBA possuem um desempenho favorável com relação a implementações que não possuem suporte a tempo real [VIN01]. Outro motivo para que CORBA esteja crescendo neste segmento é o

fato de ser a única tecnologia de objetos distribuídos que realmente está consolidada nesta área.

Além da aplicação em sistemas de tempo real e embutidos, CORBA tem compatibilidade com a arquitetura EJB de objetos distribuídos, caso o protocolo de comunicação de EJB escolhido seja o RMI-IIOP (*Remote Method Invocation over Internet Inter-ORB Protocol*)[SUN02]. CORBA também pode ser integrado a outras tecnologias, como *Web Services*, implementando-se um *gateway* SOAP-CORBA para tanto [MCH07][GOK02]. Isto pode ser útil caso se queira integrar um sistema CORBA legado a um sistema baseado em *Web Services*. Além disso, CORBA terá integração direta com SOAP em breve [OMG08].

CORBA também possui vários casos de sucesso em áreas como: astronomia (telescópio *Hubble*), aplicações militares, iluminação de pistas de aeroportos, controle de rádio de aeronaves, grandes sistemas financeiros de missão crítica, sistemas de telecomunicações, entre outros [MCH07].

2.11. Comparação entre o modelo *Ad Hoc* CORBA e modelo CORBA convencional

Como parte do estudo de implementação do algoritmo de *cache* cooperativo, foi feita uma análise da arquitetura de CORBA convencional, voltada para redes cabeadas, com a arquitetura *Ad Hoc* CORBA (Seção 2.9.2), que descreve um modelo de adaptação de CORBA para MANETs. A seguir, são apresentados vários parâmetros de comparação entre a arquitetura *Ad Hoc* CORBA e o modelo convencional de CORBA. Para cada parâmetro, são discutidas as vantagens e desvantagens referentes a cada arquitetura.

- **Desempenho - descoberta de rotas**: A arquitetura convencional de CORBA tem um melhor desempenho do que a arquitetura *Ad Hoc* CORBA porque *Ad Hoc* CORBA pesquisa a rota ao servidor do objeto requerido utilizando o algoritmo BFS, que é executado entre os ORBs presentes na rede *ad hoc*. A execução deste algoritmo desconsidera as funcionalidades contidas no protocolo de roteamento em utilização pelos nós, que se encontra na camada de rede. Isto gera uma grande redundância de operações de rede se comparado ao cenário da arquitetura convencional, onde cada nó se comunica diretamente com o servidor do objeto requerido. A redundância só é

justificada no caso de haver divergência entre a topologia física da rede e a topologia da rede "lógica" (*overlay*). A comunicação acontece de forma direta porque o protocolo de roteamento encaminha a mensagem para o destino, sem que as aplicações tenham que efetuar tal processamento.

- **Desempenho - descoberta do endereço do servidor que disponibiliza o objeto requerido**: Com relação à pesquisa do servidor que contém o objeto requerido pelo cliente pelo algoritmo BFS no *Ad Hoc* CORBA, a arquitetura convencional obtém o endereço do servidor diretamente através da sua IOR. Porém, em um ambiente dinâmico e descentralizado, talvez seja melhor ter um processamento de rede adicional e realizar uma pesquisa sobre qual elemento de rede contém o objeto requerido, em vez de ter que depender de um servidor de nomes para obter a IOR do objeto. Isto porque a implantação de um servidor de nomes em uma MANET faria com que os nós fossem obrigados a obter a IOR a partir deste servidor. De certa forma, isto contradiz a arquitetura descentralizada de MANETs, onde os nós não precisam acessar um servidor central para então se comunicar com outros nós.
- **Desempenho - conexão ponto a ponto**: Na arquitetura convencional de CORBA, cada nó da rede se comporta como um elo na conexão efetuada entre um nó cliente e um nó servidor. Para a arquitetura *Ad Hoc* CORBA, os nós que seguem esta arquitetura se comunicam com conexões do protocolo de transporte em utilização, onde o TCP seria o protocolo mais provável, inclusive entre os nós intermediários a uma conexão entre dois nós. Esse comportamento pode gerar uma complexidade de implementação e redução de desempenho considerável, se comparado ao modelo CORBA convencional. Todavia, MANETs são redes dinâmicas e talvez várias conexões TCP ponto a ponto sejam melhores do que uma conexão TCP entre origem e destino, com vários nós intermediários entrando e saindo da rede, gerando freqüentes desconexões entre os dois pontos de comunicação.
- **Conectividade**: *Ad Hoc* CORBA exige que os elementos de rede que seguirem sua arquitetura se comuniquem somente entre si. Caso haja outros nós intermediários entre dois nós *Ad Hoc* CORBA, a comunicação entre os nós *Ad Hoc* CORBA não se torna possível, a não ser que o nó intermediário esteja com um ORB em execução, mesmo que não esteja seguindo a arquitetura *Ad Hoc* CORBA. Este problema não acontece na

arquitetura convencional, pois o protocolo de roteamento da camada de rede dos nós intermediários encaminha as mensagens entre os nós que estão se comunicando.

- **Roteamento**: *Ad Hoc* CORBA utiliza o algoritmo BFS para descobrimento de rotas ao servidor do objeto requerido. O estudo da aplicação deste algoritmo no contexto de MANETs em *Ad Hoc* CORBA ainda está em andamento, onde a verificação de limites de mobilidade do ambiente onde o algoritmo será executado está sendo feita. Em contrapartida, existe muita pesquisa em andamento sobre os protocolos de roteamento no nível da camada de rede, com destaque a protocolos já bem conhecidos, como AODV (*Ad hoc On-Demand Distance Vector Routing*) e DSR (*Dynamic Source Routing*). Estes protocolos são, em sua maioria, feitos para atuarem em ambientes extremamente dinâmicos. A arquitetura convencional aproveita as funcionalidades do protocolo de roteamento em utilização, que poderia ser o AODV, DSR ou qualquer outro protocolo.
- **Atendimento a padrões**: A aplicação da arquitetura convencional procura adaptar o modelo CORBA de redes fixas ao modelo de redes *ad hoc*. Esta adaptação pode simplificar a arquitetura CORBA com relação a este tipo de rede se comparado ao modelo *Ad Hoc* CORBA, até porque boa parte das críticas à arquitetura CORBA está na sua complexidade (Seção 2.10). Por outro lado, o modelo *Ad Hoc* CORBA segue o padrão CORBA de forma mais adequada, pois a sua formulação foi feita como uma adaptação do modelo *Mobile* CORBA [OMG03] ao contexto de redes *ad hoc*.

Devido ao maior desempenho aparente, da simplicidade de implementação e da conectividade entre dispositivos mais abrangente, além da utilização de protocolos de roteamento já testados e utilizados na prática, acredita-se que a implementação do modelo convencional de CORBA no contexto de MANETs tenha resultados melhores do que a aplicação do modelo *Ad Hoc* CORBA, mesmo que o modelo *Ad Hoc* CORBA esteja mais de acordo com o padrão *Wireless* CORBA [OMG03]. O modelo *Ad Hoc* CORBA poderia ter grandes melhoras de desempenho se a descoberta de rota até o objeto requerido fosse integrada com um protocolo de roteamento que, além da descoberta de rotas, verificasse também os objetos mantidos pelos nós aos quais as rotas estão relacionadas. Existem trabalhos e estudos de integração de protocolos de roteamento com a camada de aplicação em MANETs, como pode ser visto em [TAN05] e [AUN04].

2.12. Interceptação de *Cache* e *Proxy*

Interceptação de *Cache* (ou de *Proxy*) é uma técnica popular de direcionamento de tráfego a um servidor *Proxy*, sem a necessidade de se configurar os aplicativos cliente para tanto [WES04]. A Figura 2.9 mostra um cenário de exemplo de aplicação desta técnica, onde um servidor *Proxy Squid* [SQU09] é executado em um servidor com um roteador em atividade.

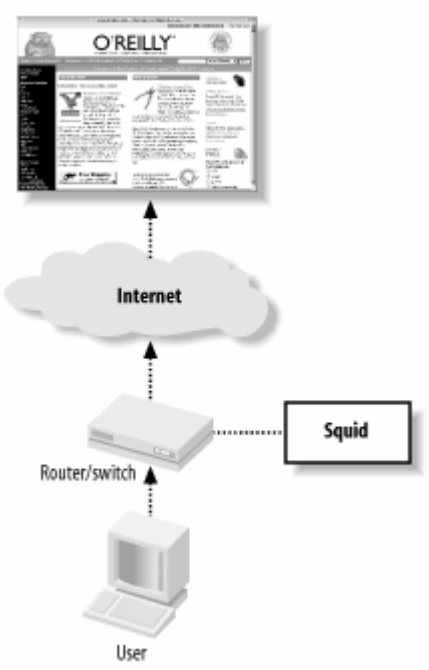

Figura 2.9: Cenário de interceptação de *cache* de uma página HTML [WES04].

Para entender como a interceptação de *cache* funciona, é descrito o seguinte cenário. Primeiro, o usuário faz uma requisição a um recurso, por exemplo, /index.html, do servidor www.oreilly.com. Após uma requisição a um servidor DNS (*Domain Name Server*) para obter o endereço IP do servidor, o navegador do usuário inicia uma conexão TCP ao servidor na porta 80, como pode ser visto na Figura 2.10

```
Packet 1
TIME:   19:54:41.320652 (0.002945)
TCP:    206.168.0.3.3897 -> 208.201.239.37.80 Syn
DATA:   <No data>
```

Figura 2.10: Início de conexão TCP ao servidor *Web* [WES04].

O *switch*/roteador recebe o pacote TCP (*Transmission Control Protocol*) SYN com destino à porta 80. O roteador está configurado para encaminhar pacotes TCP com destino à porta 80 para o servidor *Proxy*. Logo, o roteador encaminha o pacote para o servidor em utilização que, nesse exemplo, é o *Squid*. O servidor *proxy* só aceita pacotes com destino a outros endereços ou portas se o mesmo estiver com a sua interface de rede configurada para IP *forwarding*, que habilita o computador a receber pacotes endereçados a outros destinos.

Logo em seguida, o pacote é processado pelo código de filtro de pacotes em utilização pelo servidor *Proxy* (*NetFilter*, *ipchains*, etc) [WES04] [WES01]. Dependendo da regra configurada, o pacote pode ser encaminhado para o *Squid* ou encaminhado para o servidor original. Neste cenário de exemplo, o pacote é encaminhado ao *Squid*, que o aceita e envia a seguinte resposta ao cliente, como ilustrado na Figura 2.11.

```
Packet 2
TIME:   19:54:41.320735 (0.000083)
TCP:    208.201.239.37.80 -> 206.168.0.3.3897 SynAck
DATA:   <No data>
```

Figura 2.11: Resposta do *Squid* ao navegador do usuário [WES04].

Como pode ser visto na Figura 2.11, o endereço de origem é o endereço do servidor *Web*, mesmo que o pacote enviado pelo cliente não tenha alcançado o servidor. O sistema operacional simplesmente copia e troca os endereços de origem e destino do pacote SYN no pacote de resposta.

O navegador do usuário recebe o pacote SYN/ACK, estabelecendo a conexão TCP plenamente. O navegador agora acredita que está conectado ao servidor real, e envia a seguinte requisição HTTP, de acordo com a Figura 2.12.

```
Packet 3
TIME:   19:54:41.323080 (0.002345)
TCP:    206.168.0.3.3897 -> 208.201.239.37.80 Ack
DATA:   <No data>
------------------------------------------------------------------------
Packet 4
TIME:   19:54:41.323482 (0.000402)
TCP:    206.168.0.3.3897 -> 208.201.239.37.80 AckPsh
DATA:   GET / HTTP/1.0
        User-Agent: Wget/1.8.2
        Host: www.oreilly.com
        Accept: */*
        Connection: Keep-Alive
```

Figura 2.12: Envio de ACK e requisição HTTP do navegador [WES04].

O *Squid* recebe a requisição HTTP e a trata de forma normal, onde acertos de *cache* são retornados imediatamente para o navegador, enquanto que erros de *cache* (*cache misses*) são enviados para o servidor *Web*.

2.13. Resumo

Neste capítulo foram apresentados os fundamentos conceituais e tecnológicos do trabalho desta dissertação. Em redes *ad hoc*, o foco principal deste trabalho são as MANETs, que formam a base de toda a teoria apresentada nos capítulos a seguir.

Foi apresentado também o conceito geral de CORBA, assim como a estrutura de suas mensagens *Request*, *Reply* e *Fragment* do protocolo GIOP, que é o protocolo de aplicação do algoritmo proposto. A aplicação de CORBA em MANETs também foi apresentada, onde a aplicação de ORBs em diferentes ambientes de rede não altera o funcionamento da troca de mensagens do protocolo GIOP. Uma descrição sobre interceptadores portáveis de CORBA foi feita, além de uma discussão sobre vantagens e desvantagens da arquitetura CORBA e de um comparativo entre a arquitetura *Ad Hoc* CORBA e a arquitetura CORBA convencional, adaptada a MANETs. Por último, a técnica de interceptação de *cache/proxy* foi apresentada. Esta técnica serve de base para o entendimento do cenário proposto de aplicação do algoritmo de *cache* cooperativo, que é a aplicação do modelo convencional de CORBA em MANETs, discutido no Capítulo 5. No capítulo a seguir são apresentados os trabalhos relacionados ao algoritmo de *cache* cooperativo proposto, aos cenários de sua aplicação e às questões de sua implementação.

Capítulo 3

Trabalhos relacionados

Neste capítulo, são descritos trabalhos relacionados a técnicas de *cache* cooperativo empregadas na camada de aplicação do modelo de redes OSI. Primeiro, na Seção 3.1, são apresentados conceitos de *cache* de dados, *cache* de caminho de dados e de *cache* híbrido, que são amplamente utilizados por trabalhos nesta área. Na Seção 3.2, um protocolo de *cache* cooperativo aplicado a ambientes de *broadcast* de informações é apresentado, enquanto que a Seção 3.3 trata de um *middleware* de *cache* de *proxy* cooperativo, denominado MobEYE (*Mobile intercEpting proxY cachE*). Na Seção 3.4 é apresentado um trabalho sobre *cache* cooperativo voltado a uma implementação real do algoritmo descrito na Seção 3.1, descrevendo problemas encontrados na sua implementação e as soluções encontradas para resolvê-los. A Seção 3.5 apresenta parte dos poucos trabalhos existentes sobre o tratamento de *cache* e *cache* cooperativo à arquitetura CORBA, seguida da conclusão na Seção 3.6, que apresenta observações sobre os trabalhos apresentados neste capítulo.

3.1. Suporte a *Cache* Cooperativo em Redes *Ad Hoc*

3.1.1. Descrição sobre *cache* cooperativo

De acordo com [YIN04], na Figura 3.1, suponha que N_{11} é um nó fonte (central de dados) que contém um banco de dados de N itens d_1, d_2, ..., d_n. Em redes *ad hoc*, uma requisição de dados é encaminhada salto por salto, até que alcance a central de dados. Depois, a central de dados envia o dados requisitados para o nó que fez a requisição. Vários algoritmos de roteamento foram projetados para rotear mensagens em redes *ad hoc*. Para reduzir o consumo de banda e o atraso de resposta, o número de saltos entre a central de

dados e o requerente deve ser o menor possível. Mesmo que protocolos de roteamento possam ser utilizados para atingir esta meta, existe uma limitação em quanto eles podem alcançar. A seguir, três esquemas de *cache* são propostos: *cache* do caminho de dado (*CachePath*), *cache* de dado (*CacheData*) e modelo de *cache* híbrido (*HybridCache*).

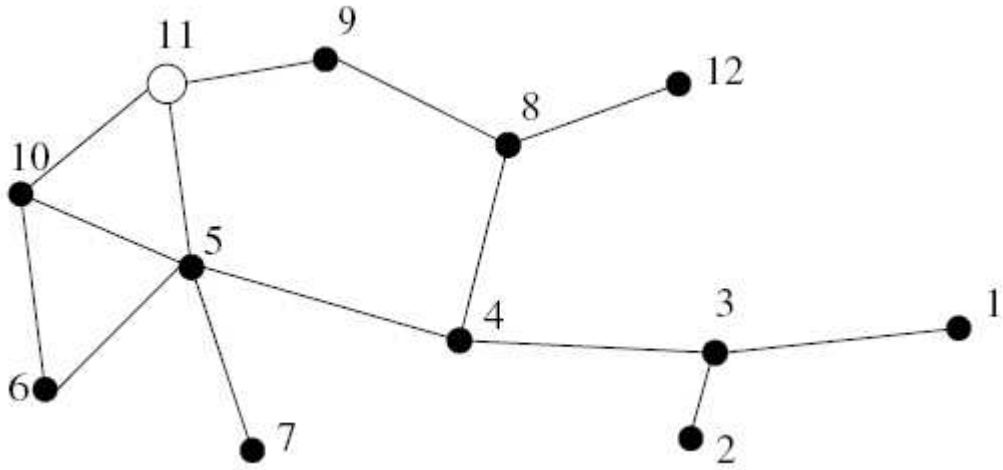

Figura 3.1: Uma rede *ad hoc* [YIN04]

3.1.2. *Cache* do caminho de dado

Suponha que o nó N_1 requisitou um item de dados d_i de N_{11}. Quando N_3 encaminha o dado d_i para N_1, N_3 sabe que N_1 tem uma cópia de d_i. Depois, se N_2 requisita d_i, N_3 sabe que a fonte de dados N_{11} está três saltos de distância, enquanto que N_1 está apenas a um salto. Logo, N_3 encaminha o pedido para N_1 em vez de N_4. Muitos protocolos de roteamento disponibilizam o número de saltos entre a fonte e o destino. Fazendo o *cache* do caminho de dados para cada item de dados, a utilização de banda e energia podem ser reduzidos, já que o dado pode ser obtido através de um número menor de saltos. Todavia, armazenando um mapa relacionando itens de dados e nós de *cache* aumenta a sobrecarga de roteamento.

Para melhorar o desempenho do problema de armazenamento citado acima, para o caminho de *cache*, um nó não precisa armazenar o caminho de informação de todos os dados passantes. Por exemplo, quando d_i é enviado de N_{11} para o nó destino N_1 pelo caminho $N_5 \rightarrow N_4 \rightarrow N_3$, N_4 e N_5 não irão fazer o *cache* da informação do caminho de d_i, já que N_4 e N_5 estão mais próximos da fonte de dados que o nó de *cache* N_1. Logo, um nó de roteamento apenas armazena o caminho de dados quando o mesmo está mais próximo do nó de *cache* que a fonte de dados.

Outro problema: por causa da mobilidade, o nó que faz o *cache* de dados pode se mover. Além disso, o dado em *cache* pode ser trocado por causa da limitação de armazenamento de *cache*. Como resultado, o nó que modificou a rota deve rotear a requisição novamente para a fonte de dados original, depois que descobrir o problema. Logo, o caminho de *cache* pode não ser confiável, e a sua utilização pode levar a problemas adicionais de sobrecarga de tráfego. Para lidar com essa situação, um nó faz o *cache* do caminho de dados apenas quando o nó de *cache* está muito próximo.

A proximidade pode ser definida pela distância da central de dados ou do nó de *cache*, pela estabilidade da rota ou pela taxa de atualização de dados. Intuitivamente, se a rede estiver relativamente estável, a freqüência de atualização de dados for baixa e a distância até o nó de *cache* for muito menor que a distância da fonte de dados, então o nó de roteamento deve fazer o *cache* do caminho do nó de *cache*.

A distância até o nó de *cache* é muito importante. Se esta distância for pequena, mesmo que o caminho estiver inválido ou o dado estiver indisponível no nó de *cache*, o problema pode ser rapidamente detectado para se resolver a sobrecarga de tráfego conseqüente. O item de caminho de *cache* pode ser armazenado em um certo nó avaliando-se a distância do nó corrente até a fonte de dados, e a distância do nó corrente até o nó que contém o dado em *cache*, sendo que esta distância deve ser menor do que a distância até a fonte de dados.

Manutenção de consistência de *cache*

Segundo [YIN04], o modelo de consistência de *cache* utilizado para as abordagens de *cache* de caminho de dados e *cache* de dados é o modelo de consistência fraca, devido às limitações de banda e energia em redes *ad hoc*.

Um modelo simples de consistência fraca de *cache* pode ser baseado no mecanismo de tempo de vida (TTL, *Time-to-live*), onde um nó considera uma cópia de *cache* como atualizada se o seu TTL não expirou, e remove o mapa de sua tabela de rotas, ou o dado em *cache*, se o TTL expirar. Como resultado, requerimentos futuros para este dado serão encaminhados para a central de dados.

Às vezes, dados inválidos podem ser úteis. Como estes foram guardados em *cache* pelo nó, isso indica que aquele nó está interessado naquele dado. Para economizar espaço, quando um item de dado em *cache* expira, o mesmo é removido do *cache*, enquanto que o seu

identificador é guardado em estado "inválido", como indicação de interesse do nó. Em [YIN04], esse controle não é feito. Quando o TTL expira, o conteúdo do *cache* é descartado.

***Cache* do dado**

Na abordagem de *cache* de dados, o nó de roteamento faz o *cache* de dados em vez do caminho, quando o mesmo percebe que o dado em questão é freqüentemente acessado. Por exemplo, na figura 5.1, se ambos N_6 e N_7 fazem uma requisição de d_i por N_5, N_5 pode detectar que d_i é um dado popular, fazendo o seu *cache* localmente. Requisições futuras por N_4 podem ser servidas por N_5 diretamente. Como a abordagem de *cache* de dados precisa de espaço extra para armazenar o dado, esta deve ser utilizada com cuidado. Suponha que uma fonte de dados receba várias requisições por d_i, encaminhadas por N_3. Os nós pelo caminho $N_3 \rightarrow N_4 \rightarrow N_5$ podem achar que d_i é um item de dados popular e que deve ser feito o *cache* de seu conteúdo. Todavia, isto irá desperdiçar uma grande quantidade de espaço de *cache* se os três nós fizerem *cache* de d_i. Para evitar essa situação, outra regra é postulada: um nó não pode fazer *cache* de dados se todas as requisições forem do mesmo nó. No exemplo anterior, todas as requisições recebidas por N_3 são de diferentes nós (N_1 e N_2). Logo, N_3 fará o *cache* dos dados. Se as requisições forem todas de N_1, N_3 não fará o *cache* de dados, mas N_1 sim. Certamente, N_5 faria *cache* de d_i, se recebesse requisições do dado a partir de N_6 e N_7.

3.1.3. Abordagem de *cache* híbrido

Cache de dados e de caminho de dados podem melhorar o desempenho do sistema significativamente. *Cache* do caminho de dados funciona melhor em situações de tamanho de *cache* pequeno, ou de freqüência de atualização de dados pequena, enquanto que *cache* de dados funciona melhor para outras situações. A aproximação de *cache* híbrido aproveita as vantagens de ambas as técnicas. Especificamente, quando um nó móvel encaminha um item de dados, o mesmo faz o *cache* do dado ou do caminho baseado em algum critério. Estes critérios incluem o tamanho do item de dados e o número de saltos que podem ser armazenados pela utilização do *cache* do caminho de dados.

Especificamente, quando um nó encaminha um item de dado, ele faz o *cache* do dado ou do caminho com base em alguns critérios. Esses critérios incluem o tamanho do item de dado (s_i), o tempo de TTL (TTL_i) e o número de saltos a salvar (H_s).

Logo, para um item de dado d_i, se s_i for pequeno, o método de *cache* de dados deve ser adotado, pois o item de dados não precisa de muito espaço do *cache*. Caso contrário, o *cache* do caminho deve ser adotado para economizar espaço de *cache*.

Porém, se o TTL_i for pequeno, o *cache* do caminho não é uma boa escolha, pois o item de dados será invalidado logo. A utilização de *cache* do caminho pode resultar na procura de um caminho inválido, resultando no re-envio da consulta para a central de dados. Logo, *cache* de dados deve ser utilizado nessa situação. Se TTL_i for grande, *cache* do caminho deve ser adotado.

Finalizando, se H_s for grande, *cache* do caminho é uma boa escolha, pois pode economizar um grande número de saltos. Caso contrário, *cache* de dados deve ser adotado para melhorar o desempenho, se tiver espaço livre suficiente no *cache*. A Tabela 3.1 a seguir mostra uma relação de fatores para os quais as técnicas de *cache* de dado e de *cache* de caminho de dado são avaliadas.

	***Cache* de dados**	***Cache* de caminho de dados**
Espaço em *cache*	Grande	Pequeno
Tamanho da rede	Grande	Pequena
Atualização de dados	Rápida	Lenta
Movimento de nós	Rápido	Lento

Tabela 3.1: Relação custo-benefício entre *cache* de dado e *cache* de caminho de dado [COO06]

3.2. Gerenciamento de *Cache* Global para *Broadcast* Móvel Não Uniforme

Broadcast de dados móveis [WU06] é considerado um modelo de disseminação de dados escalável. Em sistemas de *broadcast* móveis, objetos de dados são repetidamente enviados via *broadcast* por canais sem fio por um servidor, e um grande número de nós móveis obtém seus objetos de dados requeridos customizando o canal de *broadcast* e capturando os objetos de dados quando os mesmos aparecem. *Broadcast* móvel é útil quando existe um alto grau comum de interesse entre clientes. Exemplos de informação compatíveis para *broadcast* incluem notícias, previsão do tempo, cotação de ações, tráfego e informações

para turistas. Adicionalmente, museus podem enviar por *broadcast* vídeos de multimídia curtos sobre as exposições para dispositivos móveis de visitantes, *shopping centers* podem enviar por *broadcast* catálogos de produtos e informações sobre promoções para os clientes.

Todavia, quando o volume de dados sendo enviado por *broadcast* é grande, nós móveis têm que esperar um longo tempo até que os objetos com os dados requeridos apareçam no canal de *broadcast*. *Broadcast* não-uniforme é um mecanismo utilizado para resolver este problema no lado do servidor, com relação à comunicação *broadcast*. Neste tipo de *broadcast*, dados freqüentemente acessados são enviados por *broadcast* mais vezes que aqueles que são acessados com menor freqüência pelos nós móveis. Como a maior parte dos pedidos dos nós irá acessar dados mais populares, e estes dados serão enviados por *broadcast* mais freqüentemente, o tempo de espera médio é reduzido, ao mesmo tempo em que o tempo de espera pelos dados menos acessados fica maior.

Nesta técnica, são estudados esquemas de cooperação de *cache* para sistemas de *broadcast* não-uniformes. Primeiramente, é proposto um mecanismo simples, mas poderoso, de requisição contínua (KR, *Keep Requesting*) que reduz o tempo de espera causado por perdas de *cache* local (*cache misses*). Depois, são propostos dois novos esquemas de gerenciamento de *cache* cooperativo: gerenciamento de *cache* iniciado pela perda global de *cache* (GCM, *Global-cache-miss initiated Caching Management*) e o gerenciamento de *cache* ciente de mobilidade (MCM, *Motion-aware Caching Management*). Estes métodos consideram não apenas a freqüência de acesso, mas também a disponibilidade de dados da vizinhança do nó móvel e do canal de *broadcast*. A diferença entre os dois é que GCM tem um conhecimento mínimo dos ambientes, enquanto que MCM emprega informação de movimentação da vizinhança de nós móveis e de metadados do *broadcast*.

A proposta de [WU06] difere das propostas existentes pelos seguintes motivos: o esquema proposto requer poucas mensagens de controle, considera a disponibilidade de dados de ambos os vizinhos e do canal de *broadcast* e propõe esquemas adaptáveis ao programa de *broadcast*.

3.2.1. Modelo do sistema

No modelo proposto pelo método, apenas o servidor transmite dados para os nós, e não o contrário. O modo de *broadcast* é o não-uniforme. Dados mais populares são enviados com maior freqüência.

Um nó pode se comunicar com outro que estiver dentro do seu alcance. Existem dois tipos de comunicação entre os nós: *broadcast* ou P2P. Na comunicação *broadcast*, todos os nós que estão ao seu alcance recebem a mensagem, enquanto que, em P2P, o nó de destino recebe a mensagem enquanto que os outros nós a descartam. A transmissão *broadcast* é feita somente quando necessário, pois é uma operação que consome muita energia.

Segue um exemplo do modelo geral. Suponha que, em um tempo t_i, um nó M_q recebeu um pedido pelo objeto de dado D_k. Se D_k estiver no *cache* de M_q em t_i, então isto resulta em um acerto de *cache* local. Caso contrário, resulta em um erro de *cache* local. Quando um erro de *cache* local acontece, o nó poderá encaminhar o pedido para seus vizinhos. Os vizinhos que têm D_k irão responder a mensagem via P2P para o nó M_q. M_q irá então obter D_k do primeiro vizinho que respondeu, resultando em um acerto de *cache* global. Se M_q não receber nenhuma mensagem de resposta, então um erro de *cache* global aconteceu. Se o dado D_k for recebido por *broadcast*, então a procura por D_k resulta em um acerto de *broadcast*.

3.2.2. Requisição contínua (KR, *Keep Requesting*)

Em esquemas de *cache* cooperativo baseados em *broadcast* existentes, um erro de *cache* global resulta necessariamente em um acerto de *broadcast*. Porém, um erro de *cache* global quer dizer que apenas os vizinhos atuais não têm o dado requerido, o que não se estende aos vizinhos que estão por vir. Se algum nó entrar na rede depois, um acerto de *cache* global pode acontecer antes do acerto de *broadcast*.

A posposta quanto a isso é a de que o nó continue mandando requisições pelo dado, mesmo depois de um erro de *cache* global. O dado requerido pode ser obtido antes de um acerto de *broadcast* se um novo vizinho tem o dado requerido ou se, depois de um tempo, um dos vizinhos conseguiu obter o dado de um de seus vizinhos.

A desvantagem principal é o consumo de energia despendido por estas mensagens de *broadcast*, onde a freqüência pode ser controlada.

3.2.3. Gerenciamento de *cache* iniciado pela perda global de *cache* (GCM, *Global-cache-miss initiated Caching Management*)

GCM considera dois fatores importantes, além de incorporar KR como um componente principal: a freqüência de acesso e a disponibilidade global de objetos de dados. Freqüência de acesso ajuda a identificar objetos críticos que deveriam ser guardados em *cache*

localmente, para melhorar a taxa de acertos de *cache*. Para melhorar a disponibilidade global de dados, os objetos de dados que são guardados em *cache* por poucos nós devem ser protegidos de substituições, enquanto que os que não são freqüentemente enviados por *broadcast* devem ser protegidos.

De maneira intuitiva, dados populares que não estão disponíveis localmente devem ser guardados no *cache* local com prioridade, pois são dados acessados freqüentemente. Isto porque a obtenção freqüente dos dados a partir dos nós vizinhos, ou a partir de acertos de *broadcast*, irá gerar atrasos constantes.

No GCM, um nó não tem conhecimento do ambiente onde se encontra. Isto é feito para evitar inundações de mensagens aos nós a seu alcance, o que degrada a desempenho geral do sistema e aumenta o consumo de energia. Informações sobre freqüência de acesso e disponibilidade global de dados podem ser obtidas a partir de estatísticas locais.

O GCM melhora o tempo de resposta utilizando o mecanismo de KR, melhorando a disponibilidade global de dados. Também economiza energia, pois mensagens de controle não são utilizadas, além de reduzir o número de transmissões de objetos de dados que são facilmente obtidos do canal de *broadcast*, por sua alta freqüência de transmissão. Além disso, é adaptativo ao programa de *broadcast*, pois dados freqüentemente obtidos via *broadcast* têm menos prioridade de armazenamento local. Com GCM, cada nó é autônomo. Cada nó tem conhecimento de freqüência de acesso e disponibilidade global de dados por ele mesmo, e efetua decisões de armazenamento de *cache* independentemente.

3.2.4. Gerenciamento de *cache* ciente de mobilidade (MCM, *Motion-aware Caching Management*)

Neste modelo de gerenciamento, cada nó conhece sua localização corrente, onde o nó conhece as freqüências de *broadcast* de seus objetos de dados de interesse.

Um nó pode obter informações sobre sua localização estimando sua velocidade e direção, armazenando onde estava há alguns segundos atrás. As informações de localização podem ser obtidas de um localizador GPS.

A idéia de MCM é de postergar a substituição de *cache* e de substituir o objeto de dados que estiver disponível no canal de *broadcast*. A idéia também é a de obter dados solicitados do nó vizinho que estiver mais próximo, o que faz com que menos energia seja despendida, e também de armazenar o dado em *cache* somente um pouco antes de quando o

nó vizinho, onde o dado se localiza, sair do alcance do nó em questão, o que aumenta a disponibilidade dos dados.

Além disso, MCM melhora o tempo de resposta na melhora da disponibilidade de dados na vizinhança, fazendo o melhor uso do canal de *broadcast*. MCM é também eficiente no consumo de energia, pois obtém dados do seu vizinho mais próximo. Sua adaptação ao programa de *broadcast* é feita com a sua utilização do conhecimento de envio de *broadcast* de seus objetos de dados.

3.3. MobEYE (*Mobile intercEpting proxY cachE for MANET*)

O sistema MobEYE é um sistema de servidores *proxy* com suporte a *cache* para o acesso a páginas *web* em uma MANET [DOD04]. Em uma rede *ad hoc*, uma requisição a um nó por um arquivo é encaminhada por vários nós intermediários. MobEYE permite a localização de cópias do arquivo armazenadas em *cache* nestes nós. Opcionalmente, o arquivo pode ser procurado também na memória *cache* de nós localizados na vizinhança dos nós intermediários. Assim que a cópia é encontrada em um nó mais próximo do que o nó que mantém o arquivo original, ela é enviada ao nó que requisitou o arquivo. Cópias adicionais do arquivo também podem ser criadas. Quando um nó atua como um repetidor, fragmentos do arquivo transmitido podem ser gravados em sua memória *cache*.

Algumas premissas foram assumidas para o desenvolvimento do projeto, que são as seguintes:

- O sistema deve funcionar com qualquer tipo de protocolo de roteamento, navegador e sistema operacional, sem precisar de nenhum acesso a tabelas de roteamento ou outras informações das camadas inferiores do modelo OSI, assim como não deve precisar de nenhuma alteração nestas camadas.
- Mesmo que o sistema não esteja presente em todos os nós da MANET, a funcionalidade do sistema deve ser preservada. Isso permite a inclusão de nós não colaborativos à MANET. Todavia, se o sistema não estiver presente no nó que mantém o arquivo ou nos nós clientes, as funcionalidades do sistema não poderão ser aplicadas, pois o mesmo atua também como um servidor *Web*.
- O sistema deve ser facilmente adaptável a diferentes sistemas de arquivos distribuídos.
- Mensagens necessárias à implementação do sistema devem ser limitadas em quantidade e tamanho.

Os nós de *Proxy* nesta arquitetura possuem tratamento de interceptação de *cache/proxy*, descrito na seção 2.9. Logo, requisições de clientes são atendidas pelos nós de *proxy* de forma transparente, sem que os nós cliente percebam que o servidor que está atendendo as requisições é, na verdade, o nó de *Proxy*. Os nós de *Proxy* podem cooperar entre si, comunicando-se com o protocolo ICP (*Internet Cache Protocol*). Se um *Proxy* não mantém a cópia local de um arquivo, ele pode contatar outro nó de *Proxy* para obter a cópia. Com isto, uma rede sobreposta é formada, como pode ser visto na Figura 3.2.

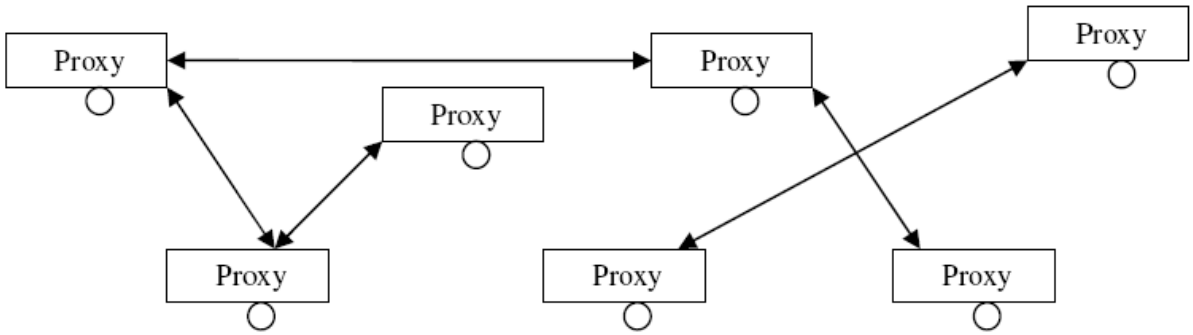

Figura 3.2: Rede sobreposta de nós de *Proxy* [DOD04].

Se um nó decide se juntar ao sistema MobEYE, ele se torna um servidor *Proxy*, juntando-se à rede sobreposta. O controle de rotas entre os nós não é reconstruído pelo MobEYE, onde a camada de rede é responsável pela funcionalidade de roteamento. Mesmo que alguns nós não estejam no sistema MobEYE, as mensagens são transmitidas pela camada de rede, sem interrupções na transmissão e encaminhamento de pacotes.

3.3.1. Interceptação de requisições e transferências de arquivos

Cada nó intermediário pertencente ao sistema MobEYE captura requisições na rede *ad hoc*. A captura é implementada utilizando-se o *software* BPF (BSD *Packet Filter*), incluso no *kernel* Linux, para disponibilizar o acesso a dados não processados de tráfego de rede a partir da interface de rede sem fio. Todos os nós do sistema MobEYE que interceptam uma requisição, e que têm o arquivo referente a requisição em *cache*, participam de uma negociação com o cliente MobEYE para decidir de onde o arquivo deve ser obtido. Todos os nós que têm a cópia em *cache* enviam uma mensagem informando que possuem o arquivo ao nó cliente, que espera pelo primeiro nó a responder (o mais próximo, ou que tenha uma

conexão mais rápida). O arquivo então é obtido deste nó, que envia a resposta HTTP para o navegador do nó cliente, como pode ser visto na Figura 3.3.

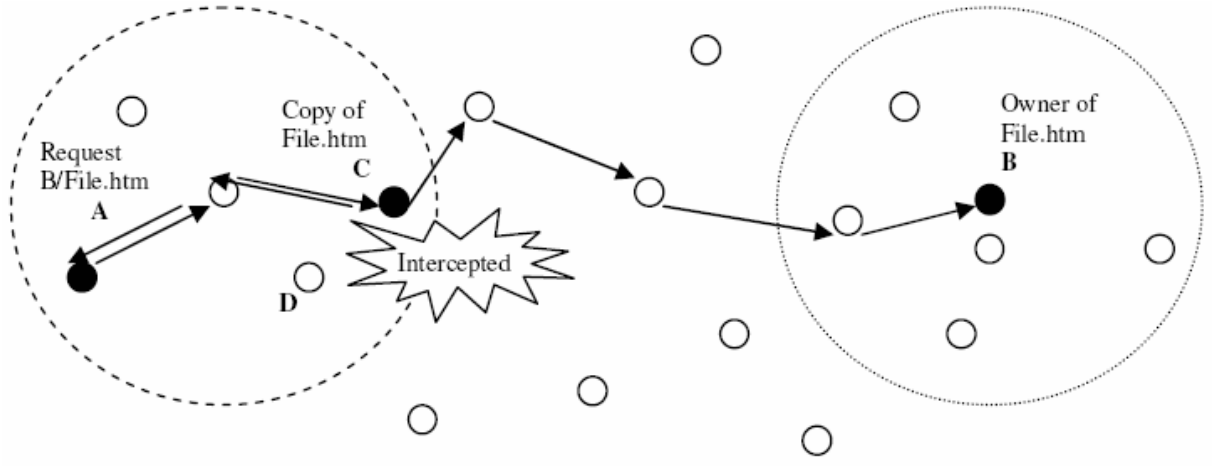

Figura 3.3: Interceptação de mensagens na arquitetura MobEYE [DOD04].

3.4. *Cache* cooperativo em redes P2P sem fio

Neste trabalho, são apresentadas uma arquitetura e uma implementação de *cache* cooperativo para redes P2P sem fio, baseadas no trabalho descrito na Seção 3.1 [ZHA08]. Através de implementações reais, são identificados problemas de arquitetura importantes com uma proposta de uma abordagem assimétrica para reduzir a sobrecarga de dados entre o espaço de usuário e o espaço de *kernel* no sistema operacional do elemento de rede. Outra contribuição do trabalho é identificar e tratar os problemas de *pipeline* de dados e interferência da camada MAC no desempenho de operações de *cache*.

O problema de desempenho identificado pelo trabalho é ilustrado pela Figura 3.4. O fato de todos os nós intermediários terem que processar todos os pacotes passantes para verificar se devem armazenar os dados em *cache* ou não acrescenta uma sobrecarga significante. Isto porque os dados dos pacotes têm que ser copiados da área de memória do *kernel* para a área de memória de usuário para o processamento de *cache* cooperativo, e depois devem ser copiados novamente da área do usuário para a área de *kernel*, para transmitir o pacote para o próximo nó pertencente à rota até o destino. Outro problema observado é o efeito de *pipeline* de dados. Na camada de transporte, se um item de dados a ser transmitido possui dados demais para serem colocados em um só pacote, os dados são colocados em vários pacotes, que são enviados para o nó de destino passando pelos nós intermediários. Como no tratamento de *cache* cooperativo os dados devem ser analisados em

cada nó, todos os pacotes devem ser recebidos pela camada de aplicação para que o item de dados seja recomposto, o que prejudica ainda mais o desempenho de transmissão.

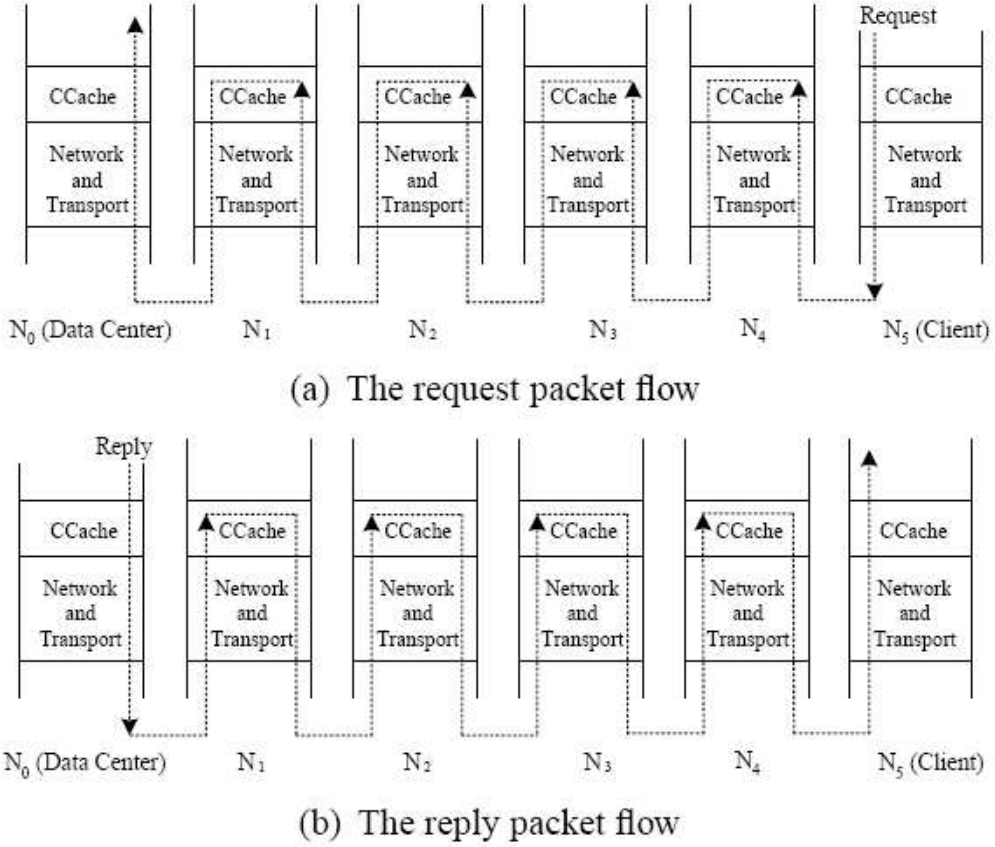

Figura 3.4: Fluxo de transmissão de pacotes em aplicações com *cache* cooperativo [ZHA08].

A resolução proposta do problema apresentado se trata do método de *cache* cooperativo assimétrico. Seu funcionamento ocorre em três fases:

1) **Encaminhamento da requisição**: Depois que uma requisição é gerada pelo nó cliente, ela é repassada para vários nós, até chegar ao nó de destino. Cada nó que faz parte da rota de transporte da requisição verifica se pode ou não fazer o *cache* dos dados requeridos, de acordo com o modelo apresentado na Seção 3.1. Caso o nó verifique que deva realizar o armazenamento em *cache*, a sua identificação é colocada em uma lista de nós candidatos ao armazenamento de dados em *cache* da resposta a esta requisição, denominada *Cache_List*, que faz parte do corpo da mensagem de requisição. Neste passo, todos os nós intermediários com tratamento de *cache* cooperativo analisam a requisição.

2) **Determinação dos nós de *cache***: o nó servidor, ao verificar os nós de *cache* contidos em *Cache_List*, determina quais nós desta lista irão efetivamente armazenar os dados da resposta à requisição em *cache*. Uma vantagem de deixar a decisão final para o servidor sobre quais nós deverão realizar o armazenamento em *cache* é a de que o servidor pode utilizar mais parâmetros para efetuar a decisão como, por exemplo, verificar a concentração dos nós solicitantes em uma mesma área para evitar o armazenamento em *cache* em muitos nós próximos uns dos outros.
3) **Encaminhamento dos dados de resposta**: ao contrário da requisição, os dados de resposta devem ser processados somente pelos nós intermediários que devam armazenar os seus dados em *cache*. Para o envio da resposta ao nó solicitante, o servidor envia a resposta diretamente ao nó de *cache* mais próximo, contido em *Cache_List*, por meio de técnicas de tunelamento [ERI94]. Quando este nó recebe o pacote de resposta, este pacote é repassado à camada de aplicação, onde se encontra o tratamento de *cache* cooperativo, que armazena os dados da resposta em *cache* e a repassa para o próximo nó de *cache* contido em *Cache_List.* Esta operação é repetida sucessivamente entre os nós de *cache* contidos em *Cache_List*, até que a resposta alcance o nó solicitante. A Figura 3.5 ilustra o fluxo do tratamento da resposta do servidor a uma requisição de um cliente.

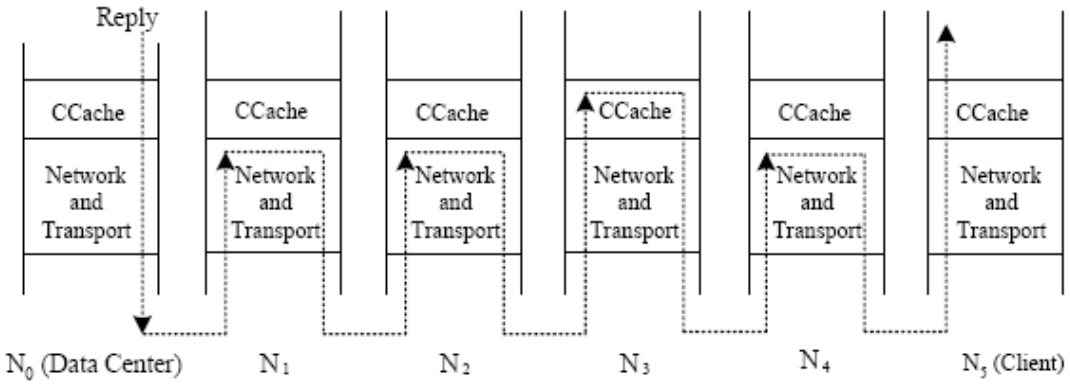

Figura 3.5: Fluxo de tratamento de pacotes no modelo de *cache* cooperativo assimétrico [ZHA08].

3.5. Trabalhos de *Cache* em CORBA

Os trabalhos existentes sobre suporte a *cache* em CORBA foram feitos com o objetivo de melhorar o desempenho de aplicações baseadas nesta tecnologia, principalmente com relação a aplicações com grande e constante tráfego de dados pela rede. Todos os trabalhos apresentados nesta seção têm em comum o tratamento de *cache* direcionado ao armazenamento de objetos inteiros em memória, onde alguns suportam também a idéia de *cache* de objetos por referência [JAY04]. Apenas o trabalho na Seção 3.4.2 aborda a questão do *cache* cooperativo e o trabalho da Seção 3.4.4 considera a aplicação de *cache* em redes móveis, mas não em MANETs.

3.5.1. CORBA *Caching*

Neste trabalho, é apresentada uma arquitetura para o tratamento de *cache* de objetos CORBA, além da proposta de soluções para os problemas de manutenção e liberação de espaço em *cache* e consistência de *cache* [TAR01]. O modelo apresentado é voltado para redes cabeadas.

3.5.2. Abordagem de *cache* cooperativo com admissão de popularidade e mecanismo de rotas

Duas questões principais são consideradas ao lidar com gerenciamento de *cache*: um mecanismo de substituição de memória em *cache* e um tratamento de consistência de *cache* [HAM01]. Como o propósito deste trabalho é lidar com *cache* de forma cooperativa, questões referentes a roteamento e propagação, ou seja, consistência de *cache*, têm que ser trabalhadas. Logo, as principais contribuições deste trabalho, que é voltado para redes cabeadas, são:

- Proposta de uma política de substituição de *cache* denominada GDSRPA (*Greedy Dual Size with Recent Popularity Admission*), que incorpora a natureza dinâmica de TTL (*Time To Live*) de objetos como um parâmetro de ordenação para substituição de *cache*. A função de ordenação utiliza parâmetros como tamanho e tempo de obtenção, assim como o custo de invalidação de *cache*.
- Um técnica de roteamento que permite que computadores que realizam tratamento de *cache* tomem conhecimento de disponibilidade de cópias de *cache* em nós vizinhos.

3.5.3. Condições de consistência para um serviço de *cache* em CORBA

Este trabalho combina visões teóricas e práticas na especificação e implementação de condições de consistência de *cache* [CHO00]. Uma definição formal de um conjunto de condições de consistência básicas é apresentada, assim como a aplicação destas condições básicas de consistência no CASCADE, que é um serviço de *cache* de objetos CORBA distribuído.

3.5.4. Invalidação de *cache* em CORBA para dispositivos sem fio

Com relação a este trabalho, um mecanismo de invalidação é proposto em conjunto com o tratamento de *cache* em CORBA para ambientes móveis, baseados no modelo *Wireless* CORBA [OMG03] [JAY04]. A abordagem armazena objetos CORBA em *cache* por valor, e é capaz de manter o objeto atualizado no nó cliente durante períodos de desconexão. Porém, a mobilidade dos clientes traz problemas ao desempenho de tratamento de *cache*, principalmente quando um nó sai de uma célula de transmissão e entra no raio de transmissão de outra célula.

3.6. Conclusão

Em 3.1, são apresentados os conceitos de caminho de *cache* de dados (*CachePath*), *cache* de dados (*CacheData*) e *cache* híbrido (*HybridCache*), que servem de base para o algoritmo de *cache* cooperativo descrito no próximo capítulo. O trabalho não discute questões reais de implementação abordando, neste aspecto, a questão de *cache* cooperativo de forma teórica. As idéias apresentadas em 3.2 sobre popularidade e disponibilidade de dados são aproveitadas no algoritmo de *cache* cooperativo proposto. No trabalho em questão, várias idéias de *cache* cooperativo são apresentadas, como o método de *Keep Requesting*, a preocupação com a disponibilidade de dados e a maior importância dada aos dados em *cache* dos nós vizinhos, com os conceitos de c*ache* local e *cache* global. Todavia, o controle de *cache* local e global envolve a formulação de um protocolo de comunicação entre os nós de *cache* que não fez parte desta dissertação, podendo ser abordado em trabalhos futuros. Questões sobre implementação em ambientes reais também não são consideradas no artigo.

Na Seção 3.3, as idéias apresentadas no trabalho sobre o *middleware* MobEYE foram aproveitadas no cenário proposto de aplicação do método de *cache* cooperativo, que é a adaptação da arquitetura convencional de CORBA em MANETs. O servidor *proxy* nos

middlewares é o que torna possível a interceptação de transmissões TCP, que é um protocolo imprescindível para a implementação de CORBA em redes *ad hoc* móveis. MobEYE, porém, trata do contexto *Web*, diferente do contexto CORBA estudado. Além disso, não há nenhuma integração adicional da camada de aplicação com a camada de rede para obtenção de dados adicionais de distância de um nó a outro, que é uma funcionalidade requerida pelo algoritmo de *cache* cooperativo proposto.

O trabalho da Seção 3.4 aborda questões interessantes de implementação, com relação aos problemas de desempenho encontrados e suas soluções. Porém, esta técnica pode não se aplicar muito bem a um modelo de *cache proxy* pois, neste modelo, o nó com tratamento de *cache* cooperativo recebe todas as mensagens endereçadas à porta TCP do servidor. Além disso, o trabalho não comenta estas questões de interceptação de pacotes TCP, relevantes em um sistema dependente de mecanismos de interceptação.

Os trabalhos descritos na Seção 3.5 mostram o que foi feito em CORBA com relação ao suporte de *cache*. Apenas um trabalho trata o cenário de mobilidade, mas não MANETs (3.5.4) e outro trabalho trata da questão de *cache* cooperativo, em redes cabeadas (3.5.2). Nenhum destes trabalhos considera o tratamento de *cache* cooperativo direcionado a CORBA em MANETs.

O capítulo a seguir fala do algoritmo de *cache* cooperativo proposto, apresentando o seu cenário genérico de aplicação e a sua arquitetura.

Capítulo 4

Modelo de *Cache* Cooperativo aplicado ao protocolo GIOP

Neste capítulo, é apresentado o algoritmo de *cache* cooperativo aplicado ao protocolo GIOP. A Seção 4.1 apresenta uma visão geral do algoritmo, mostrando também sobre o seu contexto de aplicação. A seção 4.2 apresenta o funcionamento detalhado do algoritmo, enquanto que a Seção 4.3 especifica as características que um protocolo de comunicação deve ter para a aplicação do algoritmo proposto. A Seção 4.4 trata da conclusão deste capítulo.

4.1. Descrição

Como descrito na parte inicial desta proposta (seção 1.1), a utilização da tecnologia CORBA em MANETs depende de otimizações de comunicação voltadas a este tipo de ambiente de rede. Um dos objetivos deste trabalho é utilizar as técnicas de *cache* cooperativo para ajudar neste processo, propondo um algoritmo distribuído que utilize estes conceitos. O funcionamento deste algoritmo se baseia no modelo de *cache* cooperativo híbrido, apresentado na Seção 3.1, com a implementação de idéias de popularidade e disponibilidade apresentadas na Seção 3.2.

A utilização deste algoritmo deve ser feita para armazenar dados de resposta à invocação de métodos de objetos CORBA servidores. O tratamento de consistência de *cache* do algoritmo distribuído é um tratamento de consistência fraca (Seção 2.3), baseado em tempos de TTL para que os nós de *cache* possam ter o controle de quais itens de dados em *cache* estão válidos ou não. Logo, a resposta à invocação de um método não pode ter alterações para a mesma requisição por, pelo menos, um intervalo de tempo que seja menor ou igual ao TTL estipulado para os dados armazenados em *cache* pelo algoritmo. Isto porque

não faz sentido guardar em *cache* dados de objetos que sempre enviam respostas diferentes a uma mesma requisição, pois os dados armazenados em *cache* estariam sempre defasados, perdendo o sentido de disponibilizá-los para outros nós. Este algoritmo pode ser aplicado em sistemas de disponibilização de dados persistentes, onde as mudanças de estado efetuadas nos objetos distribuídos acessados sejam menos freqüentes do que o acesso às informações disponibilizadas por estes objetos. Sistemas como a disponibilização de informações sobre a bolsa de valores, ou o controle do cadastro de funcionários de uma empresa, são exemplos de sistemas deste tipo.

Vale ressaltar como restrição que o tratamento de *cache* cooperativo aqui apresentado deve ser aplicado somente a métodos que retornam algum valor, já que o tratamento é feito nas mensagens GIOP *Reply* e *Fragment* que transportam o conteúdo do valor retornado. Além disso, a execução do método não deve alterar o estado do objeto remoto, nem de qualquer outra entidade relacionada ao seu processamento. Métodos *void* e métodos que retornam algum status de execução, além de não retornarem nenhum dado relevante a ser armazenado em *cache*, normalmente têm a função de efetuar algum processamento que altera o estado do objeto relacionado, ou de entidades envolvidas no processamento do método.

Uma diferença importante com relação ao algoritmo da Seção 3.1 é a do item de *cache* d_i. Neste algoritmo, o item de *cache* d_i é tratado como toda a resposta à invocação de um método de um objeto CORBA, mas o seu conteúdo pode ser armazenado de forma incompleta. Como as respostas à invocação de métodos podem consistir em grandes quantidades de dados, este recurso torna possível que dispositivos com menos memória possam, ao menos, armazenar parte de respostas extensas de invocações de métodos. Esta técnica é apresentada neste trabalho como fragmentação de *cache*.

O algoritmo proposto considera os conceitos de popularidade e disponibilidade de dados, para a definição de prioridade de exclusão de dados armazenados em *cache* (Seção 3.2). Com relação à disponibilidade, um certo dado, por exemplo, não deve ser descartado se nenhum dos vizinhos do nó tiver cópias do dado em questão, com o intuito de aumentar a disponibilidade deste dado, caso algum vizinho precise do mesmo no futuro. Quanto à popularidade, a idéia é a de se manter dados que sejam acessados mais freqüentemente por mais tempo.

4.1.1. Cenário básico de aplicação

Considere a situação em que, em uma MANET, um nó fonte responde a requisições de um certo nó cliente utilizando-se o tratamento de *cache* cooperativo descrito na Seção 3.1. No protocolo de transmissão proposto, os dados armazenados em *cache* são as mensagens GIOP *Reply* e *Fragment*, provenientes do nó fonte [POL08]. Estas mensagens são referentes à resposta do servidor, de acordo com a invocação de algum método específico, de algum objeto mantido pelo nó servidor. A Figura 4.1 ilustra um cenário de utilização do protocolo, onde a primeira mensagem de cada seqüência de mensagens GIOP de resposta é a mensagem GIOP *Reply*, enquanto que o resto das mensagens é do tipo GIOP *Fragment*.

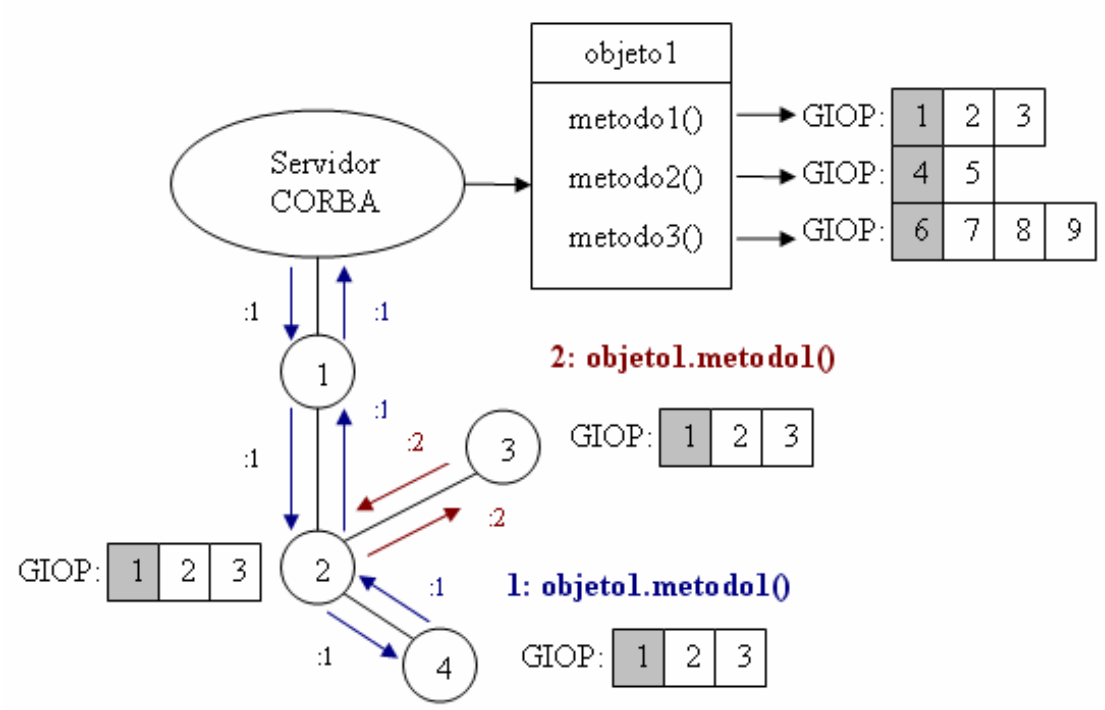

Figura 4.1: Funcionamento básico do algoritmo [POL08].

O metodo1 do objeto1, que é um objeto servidor CORBA, é executado em resposta à requisição do nó 4. O servidor executa o metodo1 do objeto1 e envia as mensagens referentes à execução para o nó 4, que são armazenadas pelo nó intermediário 2. Quando o nó 3 chama também o método1 do objeto1, sua requisição passa pelo nó 2, que analisa o pedido e verifica que o mesmo tem as mensagens GIOP referentes à resposta armazenada em seu *cache*. O nó 2, então, envia diretamente ao nó 3 a resposta para a sua requisição.

4.1.2. Visão geral do algoritmo

O funcionamento deste algoritmo se baseia no modelo de *cache* cooperativo híbrido apresentado na Seção 3.1, com exceção da definição do item de dados d_i utilizado e do tratamento de popularidade e disponibilidade de dados, como visto na Seção 3.2. A Figura 4.2 apresenta o seu algoritmo de execução.

O item de dados d_i aqui considerado consiste de uma resposta completa do servidor relativa a uma mensagem *Request*, ou seja, compreende toda a seqüência de mensagens *Reply* e *Fragment* associadas à resposta em questão. Além disso, o item de dados pode estar fragmentado, como descrito nas Seções 4.1 e 4.2.6.

O algoritmo utiliza os seguintes parâmetros de *threshold*:

- T_S: *Threshold* para o tamanho de d_i. Para o seu cálculo, de acordo com [YIN04], o ideal é utilizar a fórmula $(s_{min} + s_{máx}) * 0,4$. s_{min} e $s_{máx}$ correspondem ao valor mínimo e máximo de quantidade de memória das respostas do servidor armazenadas em *cache*.
- T_{TTL}: *Threshold* para tempo de vida de d_i. TTL_i, que é o tempo de vida de d_i, pode ser configurado para um tempo inicial curto, que pode se adaptar ao longo do tempo. Seu valor pode variar de acordo com a freqüência de modificações feitas no dado original, do qual d_i é cópia.
- T_H: *Threshold* para H_{save}. H_{save} é definido como a distância do nó de *CachePath* até o servidor CORBA, menos a distância do nó de *CachePath* ao nó que contém os dados em *cache (CacheData)*, ao qual o *CachePath* se refere. Quanto maior for o valor deste índice, maior será a vantagem de se armazenar o *CachePath*.

```
A) Quando um item de dados di é recebido:
  SE (di é o dado requerido pelo nó corrente) ENTÃO
    O item di é armazenado em cache
  SENÃO SE (tamanho de di < TS OU existe uma cópia inválida em cache
de di OU existe um cache de caminho de dados para di) ENTÃO
    O item di é armazenado em cache
  SENÃO SE (Hsave > TH) ENTÃO
    O caminho do item di é armazenado em cache
  FIM-SE
B) Quando a atualização de cache é necessária:
  ENQUANTO (NÃO houver espaço livre suficiente E existirem dados
inválidos no cache) FAÇA
    remover um dado inválido em cache
  FIM-ENQUANTO
  ENQUANTO (NÃO houver espaço livre suficiente) FAÇA
    procurar o dado válido com maior disponibilidade e menor
popularidade
    remover o dado.
  FIM-ENQUANTO
C) Quando um pedido por um item de dados chega:
  SE tiver uma cópia válida em cache ENTÃO
    enviar di para o requerente
    atualizar estatística de popularidade
  SENÃO SE (existe um caminho válido para di, no cache) ENTÃO
    encaminhar o requerimento para o nó cache
  SENÃO
    encaminhar o requerimento para a fonte de dados
  FIM-SE
```

Figura 4.2: Visão geral do algoritmo de *cache* cooperativo [POL08].

4.2. Detalhamento do algoritmo de *cache* cooperativo aplicado ao protocolo GIOP

O funcionamento detalhado do algoritmo é descrito a seguir de forma modular, podendo ser aplicado de maneira semelhante, senão igual, aos cenários de implementação descritos no capítulo 5. Este algoritmo é aplicado nos pontos de envio e recepção de dados de um elemento de rede presente em uma MANET. As operações principais do algoritmo são descritas abaixo.

- Envio de mensagens GIOP *Request*, *Reply e Fragment* relacionadas a mensagens *Reply*: Neste ponto, a rotina `verificarETratarMensagemGIOPPassante` é executada antes do envio da mensagem.
- Recepção de mensagens GIOP *Request*, *Reply* e *Fragment* relacionadas a mensagens *Reply*: Neste ponto, se a mensagem for destinada ao próprio nó, então a rotina `verificarETratarMensagemGIOPRecebida` é executada. Caso contrário, se for uma mensagem de um nó para outro, que está apenas passando pelo nó em questão, então a

rotina `verificarETratarMensagemGIOPPassante` é executada. Ambos os casos são executados antes que a mensagem seja repassada para o ORB.

A aplicação das rotinas citadas acima é feita de acordo com a Figura 4.3. Os itens destacados são as alterações a serem feitas no comportamento do elemento de rede convencional, referentes às rotinas principais do algoritmo.

```
Recepção de mensagens:

SE mensagem é para mim ENTÃO
      verificarETratarMensagemGIOPRecebida(mensagem)

      SE mensagem deve ser enviada para ORB local ENTÃO
            receber (mensagem)
      FIM-SE
SENÃO
      verificarETratarMensagemGIOPPassante(mensagem)
      SE mensagem deve ser enviada para próximo nó ENTÃO
            encaminhar (mensagem)
      FIM-SE
FIM-SE

Envio de mensagens:

verificarETratarMensagemGIOPPassante(mensagem)
SE mensagem deve ser enviada para nó destino ENTÃO
      enviar (mensagem)
FIM-SE
```

Figura 4.3: Aplicação do algoritmo de *cache* cooperativo

As rotinas `verificarETratarMensagemGIOPRecebida` e `verificarETratarMensagemGIOPPassante` se dividem em rotinas especializadas para cada tipo de mensagens suportado. Se a mensagem recebida for do tipo *Request*, então a rotina `tratarMensagemGIOPRequestRecebida` é executada. Senão, se a mensagem recebida for do tipo *Reply* ou *Fragment*, então a rotina `tratarMensagemGIOPReplyRecebida` é executada.

Analogamente, se a mensagem for uma mensagem passante do tipo *Request*, então a rotina `tratarMensagemGIOPRequestPassante` é executada. Senão, se a mensagem passante for do tipo *Reply* ou *Fragment*, então a rotina `tratarMensagemGIOPReplyPassante` é executada. A Figura 4.4 apresenta a arquitetura geral dos componentes do algoritmo. A entidade interceptadora é o ponto de interceptação de mensagens trafegando pela rede, onde as rotinas principais do algoritmo são aplicadas. O gerenciador de *cache* é utilizado para implementar as ações descritas nas próximas seções, que é a entidade principal de acesso aos itens de *CacheData* e *CachePath*, armazenados em memória.

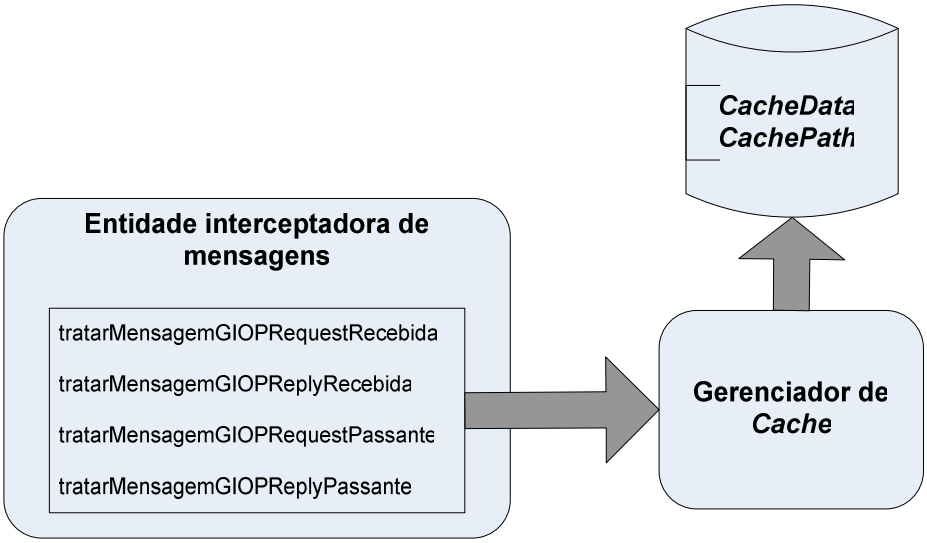

Figura 4.4: Arquitetura dos componentes do algoritmo proposto.

O funcionamento de cada uma destas rotinas, além de rotinas auxiliares e de detalhes adicionais, é descrito a seguir.

4.2.1. Funcionamento da rotina tratarMensagemGIOPRequestRecebida

Se o nó que recebeu esta mensagem não for o nó servidor dos objetos CORBA originais, então isto significa que essa mensagem de requisição foi enviada explicitamente para o nó em questão, provavelmente por meio de uma indicação de caminho de dados (*CachePath*). O nó verifica então se existe uma resposta para a requisição recebida. Caso exista, a resposta é enviada para o nó fonte que gerou a requisição. Caso não exista, a mensagem é encaminhada para algum nó que possa ter a resposta requisitada por meio de *CachePaths* armazenados, se houver algum. Caso não exista nenhum *CachePath* referente à requisição, a requisição é encaminhada ao servidor. Este caso inclusive significa que ocorreu um *Cache Miss* neste nó. A Figura 4.5 apresenta o algoritmo com este comportamento

```
tratarMensagemGIOPRequestRecebida:

SE existe CacheData para a requisição ENTÃO
      Enviar dados de CacheData ao nó solicitante
SENÃO SE existe CachePath para a requisição ENTÃO
      Encaminhar requisição ao nó de cache relacionado ao CachePath
SENÃO
      Encaminhar requisição ao servidor CORBA
FIM-SE
```

Figura 4.5: Funcionamento do algoritmo `tratarMensagemGIOPRequestRecebida`

4.2.2. Funcionamento da rotina tratarMensagemGIOPReplyRecebida

O nó verifica se a mensagem recebida é uma mensagem do tipo GIOP *Reply* ou *Fragment*. Se for, a mensagem é então armazenada em *cache* e disponibilizada para o *middleware*. Não é necessário tratar a seqüência de recebimento destas mensagens, pois o protocolo de transporte utilizado pelos ORBs, segundo a especificação CORBA, deve ser orientado a conexões e deve suportar a ordenação correta de pacotes recebidos.

4.2.3. Funcionamento da rotina tratarMensagemGIOPRequestPassante

O nó verifica se existe uma resposta para a requisição recebida. Caso exista, a resposta é enviada para o nó cliente que gerou a requisição. Se a resposta armazenada em *cache* estiver completa, ou seja, com todas as mensagens GIOP *Reply* e *Fragment* armazenadas, então a requisição recebida é bloqueada e não é enviada para o destinatário original da mensagem, pois a requisição está sendo atendida por este módulo de tratamento de *cache*, sem a necessidade de obter o resto dos dados do item de *cache*, que está completo. Caso contrário, a mensagem *Request* é encaminhada para o servidor, mas o número da última mensagem *Fragment* armazenada em *cache* é armazenada na mensagem *Request*, em seu *Service Context Id* (ver Seção 4.2.6).

Caso não existam dados em *cache* neste nó, a mensagem é encaminhada para algum nó que possa ter a resposta requisitada por meio de *CachePaths* armazenados, se houver. Se não houver nenhum *CachePath*, a requisição é enviada ao servidor CORBA. A Figura 4.6 mostra o algoritmo que descreve este comportamento.

```
tratarMensagemGIOPRequestPassante:

SE existe CacheData para requisição ENTÃO
      SE CacheData estiver completo ENTÃO
            Enviar dados de CacheData para o nó solicitante
      SENÃO
            Informar número da última mensagem Fragment armazenada em
            cache
            Encaminhar requisição ao servidor CORBA
      FIM-SE
SENÃO SE existe CachePath para requisição ENTÃO
      Encaminhar requisição ao nó de cache relacionado ao CachePath
SENÃO
      Encaminhar requisição ao servidor CORBA
FIM-SE
```

Figura 4.6: Funcionamento do algoritmo `tratarMensagemGIOPRequestPassante`

4.2.4. Funcionamento da rotina tratarMensagemGIOPReplyPassante

O nó verifica se a mensagem recebida é uma mensagem do tipo GIOP *Reply* ou *Fragment*. Caso seja, o *object key* presente no *service context* da mensagem é obtido para identificar se existe algum dado armazenado em *cache* para o *object key* em questão (ver Seção 4.2.5). Se existir, considerando-se d_i como o dado em *cache* e de acordo com os parâmetros de *threshold* apresentados na Seção 4.1.3:

- Se o tamanho de d_i for menor do que T_S ou se existe uma cópia defasada em *cache* de d_i ou se existe um *CachePath* armazenado para d_i, então d_i é armazenado em *cache* e, caso exista um *CachePath* associado, o dado em *CachePath* é atualizado.
- Caso contrário, se H_{SAVE} for maior do que T_H e o tempo de TTL de d_i for maior do que T_{TTL}, então o *CachePath* de d_i é armazenado, como sendo o caminho para o nó que deve receber este dado como resposta.

Caso o dado a ser armazenado seja uma mensagem GIOP *Fragment*, então o dado é armazenado dentro do item de *cache* como uma das mensagens da cadeia de dados, no final da cadeia de mensagens armazenada. A Figura 4.7 mostra o algoritmo que descreve este comportamento.

```
tratarMensagemGIOPReplyPassante:

SE (tamanho de d_i < T_S OU existe uma cópia inválida em cache de d_i OU
existe um CachePath para d_i) ENTÃO

      Armazenar d_i em cache
      SE existir um CachePath para d_i ENTÃO
            Atualizar CachePath de d_i
      FIM-SE

SENÃO SE (H_save > T_H E TTL_i > T_TTL) ENTÃO
    Armazenar CachePath de d_i
FIM-SE
```

Figura 4.7: Funcionamento do algoritmo `tratarMensagemGIOPReplyPassante`

4.2.5. Inclusão de atributos no *Service Context*

Para a implementação deste algoritmo, alguns dados tiveram que ser incluídos no *Service Context* das mensagens GIOP *Request* e *Reply*. Estas informações são as seguintes:

- ***Timestamp***: Referente à data e hora de criação da mensagem *Reply*. Este atributo é utilizado para os cálculos de TTL de cada mensagem de resposta. O valor do seu campo é de 64 bits, representando um *timestamp* padrão UNIX em milissegundos. Pode ser representado em uma IDL pelo tipo *long long*.
- ***Fragment***: Informa o número da última mensagem transmitida para um nó cliente. Esta informação é colocada no *service context* de mensagens *Request* (ver Seção 4.2.6). O seu valor é representado por um número inteiro de 32 bits, que pode ser representado em uma IDL pelo tipo *long*.
- ***Request Key***: Especifica o par de informações *object_key* (até GIOP 1.1) ou *target* (a partir de GIOP 1.2), e *operation*, que identificam o objeto e o método que foram solicitados pelo cliente. Estas informações são informadas em mensagens GIOP *Reply*, para que seja possível identificar de qual objeto e método o *Reply* é uma resposta, pois estas informações não se encontram no cabeçalho da mensagem GIOP *Reply*. Seu valor pode ser representado em uma IDL como uma seqüência variável de *bytes* (tipo *octet*).

4.2.6. Tratamento de fragmentação de *cache*

Nos trabalhos feitos sobre *cache* cooperativo apresentados no capítulo 3, para um item de *cache* convencional assume-se que todos os dados que o compõem estejam presentes no seu conteúdo. O problema desta abordagem é que, se um item de *cache* tiver muitos dados, o seu armazenamento pode exigir uma quantidade muito grande de memória. Como conseqüência, isso pode fazer com que nós de *cache* deixem de armazenar itens de *cache* pela grande quantidade de memória que estes itens podem requerer. Na fragmentação de *cache*, o item de dados pode ter parte dos seus dados armazenada em memória.

O conceito de fragmentação de *cache* se aplica bem ao contexto de GIOP, pela existência de mensagens GIOP *Fragment* como fragmentos de resposta da mensagem GIOP *Reply*. Logo, cada mensagem *Fragment* pode ser considerada como um fragmento de toda uma cadeia de mensagens GIOP *Reply* e *Fragment*, integrantes de um item de dados em *cache*. Os pontos principais do algoritmo onde a fragmentação de *cache* é aplicada são os seguintes:

- Na operação de armazenamento de dados em *cache*, se o tamanho do item de dados a ser armazenado em *cache* é maior do que T_S, ou maior do que o espaço disponível

para armazenamento no nó de *cache*, então são armazenados os fragmentos do item de dados que puderem ser salvos em memória.

- Na exclusão de dados em *cache* para a manutenção de memória, o item de dados não é excluído por completo. Em vez disso, cada fragmento do item de dados é excluído, até que seja liberado espaço em memória para o armazenamento de mais dados.
- Na execução da rotina `tratarMensagemGIOPRequestPassante`, caso o item de dados em *cache* que estiver sendo transmitido para o nó cliente esteja fragmentado, o nó de *cache* encaminha o *Request* do cliente para o servidor, afim de que algum nó de *cache* que esteja no caminho possa servir o resto da requisição que foi atendida pelo nó de *cache* anterior, enviando o resto da resposta para o nó cliente. A informação sobre qual foi o último fragmento transmitido para o cliente é colocada no *Service Context* da mensagem *Request* (ver Seção 4.2.5). Caso nenhum nó de *cache* intermediário continue a transmissão para o cliente, o nó que tem o objeto CORBA servidor envia as mensagens faltantes, caso o TTL da resposta não tenha expirado. Caso contrário, toda a resposta é transmitida para o nó cliente.

A Figura 4.8 descreve o funcionamento do tratamento de fragmentação de *cache*.

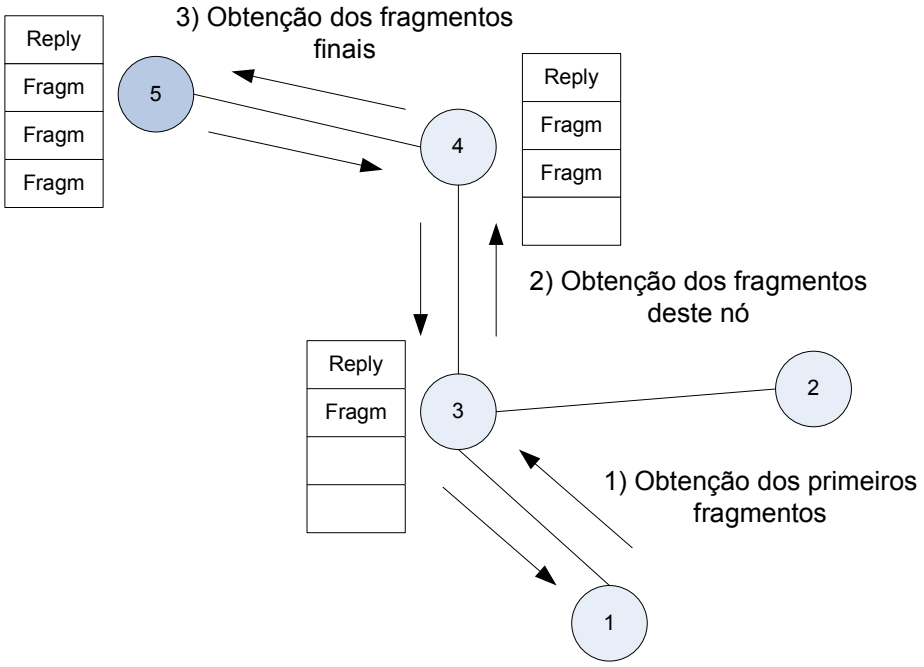

Figura 4.8: Funcionamento do tratamento de fragmentação de *cache*.

4.2.7. Rotinas auxiliares

Para a implementação das rotinas principais do algoritmo, algumas rotinas auxiliares são necessárias. Estas rotinas estão relacionadas ao armazenamento de novos dados em *cache*, além da exclusão de dados e manutenção da memória *cache*, que são descritas a seguir.

- **Armazenamento de dados em *cache*:** Para as rotinas que armazenam dados em *cache*, se for uma mensagem do tipo *Reply*, é feita uma verificação de espaço livre para armazenamento da mensagem. Depois disto, o horário de criação da mensagem, armazenado como um *Service Context*, é extraído da mensagem e armazenado no dado de *cache* referente à mensagem, que é armazenado em *cache*. Caso a mensagem seja do tipo *Fragment*, também é feita uma verificação de espaço livre para armazenamento da mensagem e, se houver um dado em *cache* referente ao *Reply* anteriormente recebido, este dado não foi recebido anteriormente e o dado em *cache* está incompleto, ou seja, nem todas as mensagens *Fragment* foram recebidas, então a mensagem é armazenada com relação ao dado em *cache* já armazenado em memória.
- **Exclusão de dados em *cache*:** Quando a remoção de dados da memória *cache* nos nós com módulos de *cache* cooperativo deve ser realizada, devido à falta de espaço de armazenamento, o descarte é feito de acordo com os seguintes critérios, em ordem de importância:
  1) Fragmentação do dado (se está completo ou não).
  2) Validade do dado, verificando se o seu TTL não expirou.
  3) Popularidade.
  4) Disponibilidade.
  5) Tempo restante de TTL dos dados em *cache* a serem excluídos.

  A popularidade é atualizada através de um contador de mensagens de resposta enviadas pelo nó de *cache*, associado ao item de *cache* em questão. Já a disponibilidade é calculada de acordo com o número de itens de *CachePath* associados a um mesmo *CacheData*. Quanto mais itens de *CachePath* um item de dados em *cache* tiver, maior será a sua disponibilidade.

4.3. Aplicação do algoritmo em outros protocolos de comunicação

Apesar do algoritmo de *cache* cooperativo proposto neste trabalho ser aplicado especificamente ao protocolo GIOP, este algoritmo poderia ser aplicado a outros protocolos de comunicação similares. As características que um protocolo deve ter para que o algoritmo proposto possa ser aplicado são as seguintes.

- Devem existir mensagens de requisição e de resposta à requisição em questão, referentes a um certo serviço oferecido por um nó.
- Nas mensagens de requisição e de resposta, caso as informações descritas na seção 4.2.5 não estejam presentes, é necessário que exista um atributo de dados variável para a passagem de parâmetros adicionais específicos de controle do algoritmo de *cache* cooperativo. Como visto anteriormente, no protocolo GIOP, o campo *service context* tem esse papel, onde vários parâmetros são colocados para possibilitar o controle de mensagens GIOP *Request* e *Reply* pelos ORBs. Caso não exista um campo de dados variável, deve existir ao menos uma área reservada na especificação da mensagem de requisição e de resposta do protocolo, que seja suficiente para armazenar os dados descritos na Seção 4.2.5. Com relação ao campo *REQUEST_KEY*, presente no *service context* referente ao algoritmo de *cache* cooperativo, o seu valor deve ser um indicativo de qual requisição uma mensagem de resposta se refere. O seu valor não precisa estar necessariamente relacionado ao método de um objeto, pois este é um valor específico do contexto de CORBA.
- O protocolo pode ter a mensagem de resposta fragmentada em várias outras mensagens, formando uma cadeia de mensagens de resposta. Isto pode ocorrer caso o conteúdo dos dados de resposta a serem transmitidos pelo servidor tenha um tamanho superior à capacidade de transmissão da mensagem de resposta.
- Por fim, o protocolo deve ter alguma informação na mensagem de resposta, ou em uma das mensagens referentes à extensão do seu conteúdo, que indique qual é a última mensagem da cadeia de mensagens que compõem a resposta em questão.

O algoritmo proposto poderia assim ser aplicado, por exemplo, a protocolos de comunicação voltados à transmissão de mídia, ou de fluxo contínuo (*streaming*) de dados. A fragmentação de *cache* é um recurso útil para o controle de *cache* nestes protocolos.

4.4. Conclusão

Neste capítulo foi apresentado o algoritmo de *cache* cooperativo aplicado ao protocolo GIOP, apresentando o contexto geral de sua aplicação e os detalhes do seu funcionamento, bem como as características que um protocolo de comunicação deve ter para que este algoritmo possa ser utilizado, para tratamento de *cache* cooperativo no protocolo.

As vantagens da arquitetura deste algoritmo, além da otimização de desempenho nas transmissões entre os ORBs, está no tratamento de popularidade, disponibilidade e da fragmentação de *cache*, que são conceitos que visam uma melhoria ainda maior no desempenho de transmissões GIOP. Popularidade e disponibilidade são conceitos apresentados em [WU06], que foram adaptados ao algoritmo proposto. Quanto ao conceito de fragmentação de *cache*, esta foi uma idéia concebida durante a realização deste trabalho. Vale ressaltar que os atributos de *Service Context* facilitaram muito o tratamento do algoritmo sobre o protocolo GIOP, evitando o encapsulamento das mensagens GIOP em outro tipo de mensagens, o que tornaria obrigatória a elaboração de um novo protocolo apenas para a transmissão de dados de controle do algoritmo. Por outro lado, a ausência de formulação de um protocolo para comunicação entre os nós de *cache* restringiu as ações de *cache* cooperativo tomadas por estes nós, que têm como base apenas as mensagens em tráfego sobre suas interfaces de rede para tanto. Um protocolo de comunicação entre estes nós poderia ser feito para que pudessem se comunicar entre si disponibilizando, por exemplo, informações sobre quais itens de *cache* cada um possui. *CachePath* é uma abordagem interessante para roteamento de dados em c*ache* entre nós vizinhos, mas um protocolo de comunicação que torne possível a troca de informações entre estes nós, mesmo gerando um pouco mais de tráfego na rede como conseqüência, é mais adequada, principalmente se a mobilidade dos nós for alta.

O próximo capítulo descreve como o algoritmo apresentado neste capítulo poderia ser implementado para o tratamento de mensagens GIOP em uma MANET, além de uma apresentação dos dados obtidos de sua simulação e da implementação da simulação.

Capítulo 5

Implementação e análise de desempenho

Neste capítulo, é apresentado o cenário previsto de implementação do algoritmo de *cache* cooperativo aplicado ao protocolo GIOP, proposto no capítulo 4, assim como a implementação da simulação realizada e os seus resultados. A Seção 5.1 descreve o modelo de implementação do algoritmo em MANETs. A Seção 5.2 descreve a simulação, apresentando a sua implementação, o ambiente elaborado e os resultados de sua execução. Por fim, a Seção 5.3 apresenta a conclusão deste capítulo.

5.1. Implementação real do algoritmo de *cache* cooperativo

A implementação do algoritmo é feita em um módulo de servidor *proxy* para mensagens GIOP, transmitidas pelo ORB local. A interceptação de mensagens GIOP entre os nós da MANET deverá ocorrer de forma similar ao modelo apresentado em [DOD04]. Os nós que possuem tratamento de *cache* cooperativo têm a funcionalidade de um servidor *Proxy* para o protocolo GIOP, onde é implementado o suporte a *cache* cooperativo para o protocolo GIOP. Isto permite a interceptação de pacotes TCP/IP por parte do servidor *Proxy*, que faz com que o ORB local acredite que está se comunicando diretamente com um ORB remoto. A técnica que faz com que isso seja possível é chamada de interceptação de c*ache* (*Interception Caching*), descrita no capítulo 2. A Figura 5.1 ilustra o cenário de aplicação do tratamento de *cache* cooperativo para o protocolo GIOP em uma MANET. O ORB ilustrado na Figura 5.1 pode ser tanto um ORB da arquitetura *Ad Hoc* CORBA quanto um ORB da arquitetura CORBA convencional, adaptado ao contexto de MANETs.

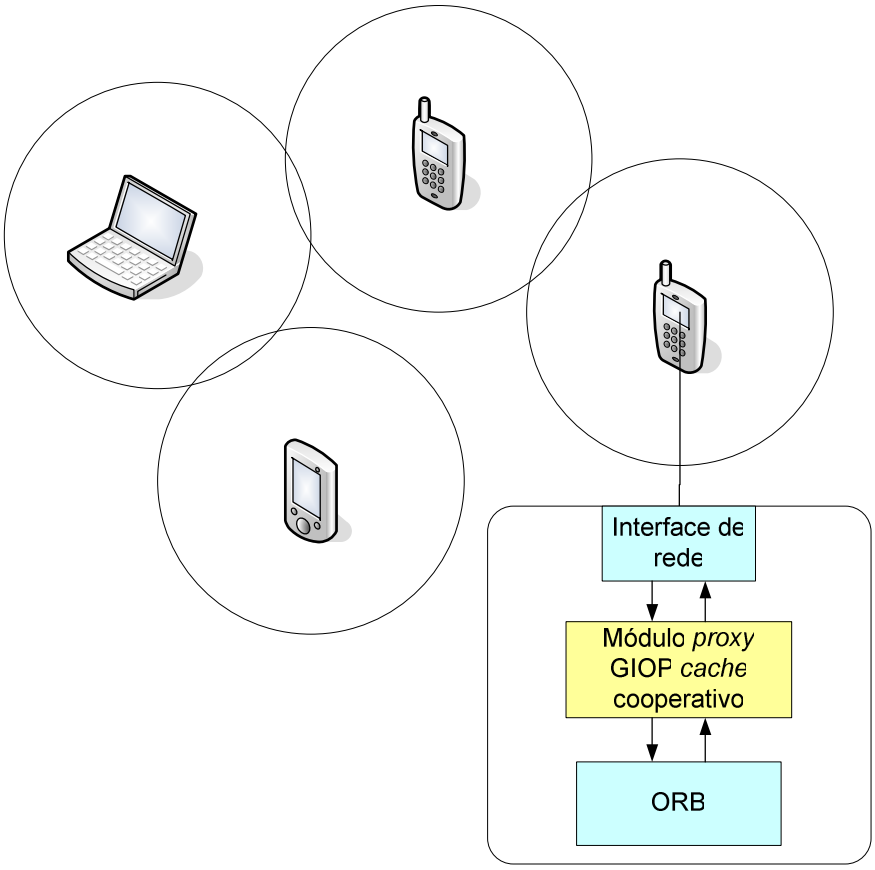

Figura 5.1: Módulo de *Proxy* de *cache* cooperativo para o protocolo GIOP em MANETs.

O algoritmo não pode ser implementado utilizando-se *portable interceptors* (Seção 2.9.3) de CORBA por causa do tratamento direto que o algoritmo faz com relação a mensagens GIOP *Fragment*, além do seu suporte a fragmentação de *cache*. A aplicação do algoritmo em eventos de *portable interceptors* como *send_request*, *receive_request*, *send_reply* e *receive_reply* permite que mensagens GIOP *Request* ou GIOP *Reply* possam ser acessadas diretamente, além da possibilidade de modificação de informações de *service context* em ambas as mensagens. Porém, não há indicações de que seja possível verificar cada mensagem de resposta recebida por vez, que fazem parte da cadeia de mensagens de resposta recebidas (mensagem GIOP *Reply* seguida de mensagens GIOP *Fragment*). Isto impossibilita a operação direta do algoritmo sobre mensagens GIOP *Fragment*, tornando inviável a funcionalidade de *cache* fragmentado de dados. No modelo *Ad Hoc* CORBA, a implementação do algoritmo seria possível inserindo o seu código do algoritmo nos pontos de interceptação de uma requisição CORBA, mas o suporte a fragmentação de *cache* não seria possível.

Para a interceptação de pacotes, APIs como *ipchains (kernel* do Linux até versão 2.4), *iptables* (*kernel* do Linux da versão 2.4 em diante), *ipfw* (*FreeBSD*), *pf* (*OpenBSD*) ou *IPFilter* (*NetBSD*, *Solaris* e outros sistemas operacionais variantes de BSD) podem ser utilizados [WES01][WES04]. A Seção 2.11 explica como esta interceptação funciona. A interceptação do pacote faz com que este seja encaminhado para um endereço IP e porta TCP ou UDP. No caso desta implementação, o pacote seria encaminhado para o endereço local da própria máquina onde o pacote foi interceptado (127.0.0.1), para a porta onde o servidor *proxy* de *cache* cooperativo está esperando por mensagens GIOP.

O algoritmo de *cache* cooperativo para o módulo de *Proxy* de mensagens GIOP é descrito na Figura 5.2.

```
Recepção de mensagens:

SE mensagem é para o ORB local ENTÃO
      verificarETratarMensagemGIOPRecebida(mensagem)

      SE mensagem deve ser enviada para ORB local ENTÃO
            receber (mensagem)
      FIM-SE
SENÃO
      verificarETratarMensagemGIOPPassante(mensagem)
      SE mensagem deve ser enviada para próximo nó ENTÃO
            encaminhar (mensagem)
      FIM-SE
FIM-SE


Envio de mensagens:

verificarETratarMensagemGIOPPassante(mensagem)
SE mensagem deve ser enviada para nó destino ENTÃO
      enviar (mensagem)
FIM-SE
```

Figura 5.2: Funcionamento geral do algoritmo de *cache* cooperativo no contexto de *Ad Hoc* CORBA.

Para a recepção de mensagens, se a mensagem for endereçada ao ORB local, então a rotina `verificarETratarMensagemGIOPRecebida` é executada. Em seguida, caso a mensagem deva ser enviada para o ORB local, ela é enviada. Caso contrário, não é enviada, pois a execução da rotina de *cache* cooperativo pode ter atendido a uma requisição do cliente, caso a mensagem recebida seja uma mensagem GIOP *Request*. Se a mensagem não for endereçada ao ORB local, então o ORB local está operando como um nó intermediário para

esta mensagem, e a rotina `verificarETratarMensagemGIOPRecebida` é executada. Em seguida, se a mensagem deve ser enviada para o seu destino, ela é enviada. Caso contrário, o seu envio é bloqueado.

Para o envio de mensagens, a rotina `verificarETratarMensagemGIOPPassante` é executada. Analogamente à recepção de mensagens, se a mensagem deve ser enviada ao seu destino, ela é enviada. Caso contrário, o seu envio é bloqueado.

5.2. Simulação

A simulação foi feita utilizando-se o simulador SWANS (*Scalable Wireless Ad hoc Network Simulator*) [JIS04]. A aplicação do algoritmo de *cache* cooperativo foi feita na camada de rede dos nós pertencentes ao cenário de simulação, utilizando-se o protocolo UDP como protocolo de transporte. Este cenário foi escolhido por causa da facilidade e flexibilidade de implementação da simulação, se comparada com os cenários reais. Isto porque o acesso às mensagens passantes é direto, pois todas as mensagens que trafegam pelo nó passam pela camada de rede. Com isso, considerando o caso da arquitetura CORBA convencional, não é necessária a modificação do código do simulador para interceptar mensagens passantes que não são destinadas ao próprio nó, ou a implementação de um módulo servidor em uma *thread* separada para a análise constante de mensagens interceptadas. Já com relação a *Ad Hoc* CORBA, não é necessária a implementação do algoritmo BFS, ou a transmissão das mensagens GIOP de nó a nó por conexões TCP/IP.

Todas as mensagens GIOP *Request*, *Reply* e *Fragment* são interceptadas no nível da camada de rede do modelo OSI, onde se encontra o tratamento de *cache* cooperativo. Apesar da facilidade de acesso ao tráfego de mensagens, vale ressaltar que a aplicação do algoritmo nesse contexto, embora esteja próximo do comportamento real de uma implementação como será demonstrado a seguir, serve apenas para fins de simulação, pois este tipo de implementação é inviável pelos seguintes motivos [ZHA08].

- A camada de rede tem a sua implementação feita no nível de *kernel*, o que implica em uma aplicação de *cache* cooperativo no nível do *kernel*. É difícil customizar uma implementação neste nível, assim como é difícil para uma implementação deste tipo tratar de requisitos de diferentes aplicações.

- A implementação no nível de *kernel* exige um maior consumo de memória por causa do uso de *CacheData*, pois os dados em *cache* teriam que ser armazenados no espaço de memória do *kernel*.
- Não existe um protocolo de roteamento padronizado nos dias de hoje. A implementação de *cache* cooperativo na camada de rede demanda que os módulos de *cache* e de roteamento estejam fortemente acoplados, além de que o módulo de roteamento tem que ser modificado para que funcionalidades de *cache* sejam adicionadas. Além disso, integrar o módulo de *cache* cooperativo com diferentes protocolos de roteamento envolveria um esforço enorme.

A seguir são apresentadas as restrições do ambiente da simulação, em comparação com os cenários reais de aplicação do algoritmo de *cache* cooperativo.

5.2.1. Restrições

Como a interceptação de mensagens é feita na camada de rede, as ações de interceptação são feitas de forma mais simples e rápida, se comparada aos cenários reais. Logo, as restrições das simulações com relação a cada cenário real de aplicação do algoritmo são as seguintes:

- **Arquitetura CORBA convencional**: para esta arquitetura, é necessária a modificação do código do simulador para interceptar mensagens passantes que não são destinadas ao próprio nó, além da implementação de um módulo servidor em uma *thread* separada para a análise constante de mensagens interceptadas. Como os nós que tem suporte a *cache* cooperativo têm um servidor *proxy* para o protocolo GIOP em execução, o mesmo também teria que ser implementado na simulação.
- ***Ad Hoc* CORBA**: seriam necessárias a implementação do algoritmo BFS e a transmissão das mensagens GIOP de nó a nó por conexões TCP/IP. A implementação de código Java no SWANS segue o modelo de tempo de simulação (*simulation time*) [BAR04]. Segundo este modelo, conexões criadas em um código Java são consideradas entidades de simulação, que têm suas operações escalonadas pelo *kernel* do SWANS na execução da simulação. Como cada método de uma conexão é considerado um evento a ser escalonado, este modelo não permite, por exemplo, o envio de mensagens por uma conexão se a mesma *thread* de execução estiver

processando dados provenientes de uma outra conexão já aberta. Seria necessário processar os dados da conexão aberta até o final do método para depois enviar as mensagens pela outra conexão. Detalhes como este dificultam a implementação de cenários de transmissão de mensagens mais complexos, que é o caso da simulação desta arquitetura. Além disso, o servidor de *proxy* do protocolo GIOP também teria que ser implementado neste cenário para os nós com suporte a *cache* cooperativo.

Além das restrições apresentadas acima, o protocolo UDP teve que ser utilizado nas simulações. Isto porque o SWANS possui problemas de implementação por ser um simulador relativamente novo e sem um grande suporte por parte da comunidade científica, o que afeta inclusive a sua implementação do protocolo TCP que, mesmo com algumas tentativas de correção de código, apresentou muitos problemas de simulação. Devido a esse problema, a simulação foi feita da seguinte forma:

- Como é possível garantir a seqüência correta de mensagens no SWANS pelo tempo de simulação [BAR04], não existem problemas de ordenação de mensagens nas simulações, mas existem problemas de recepção de mensagens, pois algumas mensagens não são recebidas pelos nós por causa da mobilidade da rede, atrasos de sinal de rádio, etc. Para resolver este problema, ao receber as respostas das requisições para os objetos remotos, cada nó verifica se a resposta obtida é igual a resposta esperada a partir da invocação remota do método. Se a resposta não for a mesma, a requisição é feita novamente. Em um ambiente com TCP isso não é necessário, pois o próprio protocolo garante que os dados recebidos pelo nó cliente são válidos e ordena as mensagens recebidas.
- Para as medições de desempenho, os resultados considerados realmente importantes como contribuição foram as comparações realizadas entre cenários diferentes de execução, onde os resultados são referentes às diferenças de desempenho entre os cenários simulados. Dados como tempos de transmissão isolados de apenas um cenário não podem ser considerados válidos, pois são dados que se referem a um cenário que não existe na prática, e que não pode ser comparado com dados de cenários reais de implantação de ORBs em MANETs.

5.2.2. Vantagens de utilização do SWANS e da implementação da simulação

Mesmo com os problemas de implementação citados na seção anterior, o SWANS é um simulador com um grande potencial por fazer com que programas codificados em Java possam ser simulados em MANETs. Isso possibilitou a construção do algoritmo de *cache* cooperativo em Java, que é um código que pode ser reaproveitado em implementações futuras de cenários reais de aplicação. Além disso, a arquitetura do algoritmo foi feita de forma modular. Isto porque, além de ser uma prática fundamental de desenvolvimento de *software*, a modularização do algoritmo permite que o mesmo possa ser reaproveitado em diferentes cenários de aplicação, sem que haja grandes mudanças de adaptação do seu código ao cenário em questão.

5.2.3. Implementação

A versão do SWANS utilizada para as simulações foi a versão 1.0.6 [JIS04]. Existe uma ramificação do SWANS chamada SWANS++ [SWA07], onde novas funcionalidades e correções de *bugs* são disponibilizadas. Porém, não foi possível nem mesmo executar simulações simples com este simulador, sem grandes esclarecimentos sobre estes problemas em seu fórum na *Internet*.

A aplicação do código foi feita na classe `jist.swans.net.NetIp` do SWANS, responsável pelo tratamento do protocolo IP, na camada de rede. De forma similar à aplicação do algoritmo em cenários reais, apresentada nas seções anteriores, a aplicação do algoritmo nessa classe foi feita como mostra a Figura 5.3.

```
Método receive:

SE mensagem é para mim ENTÃO
      verificarETratarMensagemGIOPRecebida(mensagem)

      SE mensagem deve ser enviada para ORB local ENTÃO
            receber (mensagem)
      FIM-SE
SENÃO
      verificarETratarMensagemGIOPPassante(mensagem)
      SE mensagem deve ser enviada para próximo nó ENTÃO
            encaminhar (mensagem)
      FIM-SE
FIM-SE

Método send:

verificarETratarMensagemGIOPPassante (mensagem)
SE mensagem deve ser enviada para nó destino ENTÃO
      enviar (mensagem)
FIM-SE
```

Figura 5.3: Aplicação do algoritmo de *cache* cooperativo na implementação das simulações.

Uma ilustração da arquitetura dos módulos de gerenciamento de dados de *CacheData* e *CachePath* é feita na Figura 5.4. As classes principais da implementação, que são referenciadas pela classe `jist.swans.net.NetIp`, são as classes *CacheDataController* e *CachePathController*. Estas classes possuem métodos que são executados pelas rotinas `tratarMensagemGIOPRequestRecebida`, `tratarMensagemGIOPReplyRecebida`, `tratarMensagemGIOPRequestPassante` e `tratarMensagemGIOPReplyPassante`. A função destas classes é armazenar os dados de *cache* da aplicação e prover métodos de controle sobre estes dados. Os dados em *cache*, que são armazenados por estas classes, são objetos das classes *CacheData* e *CachePath*, ambos derivados da classe *Cache*. A classe *RemovableCacheItem* relaciona os dados em *cache* que devem ser excluídos, para a manutenção do armazenamento em memória destinado a estes dados.

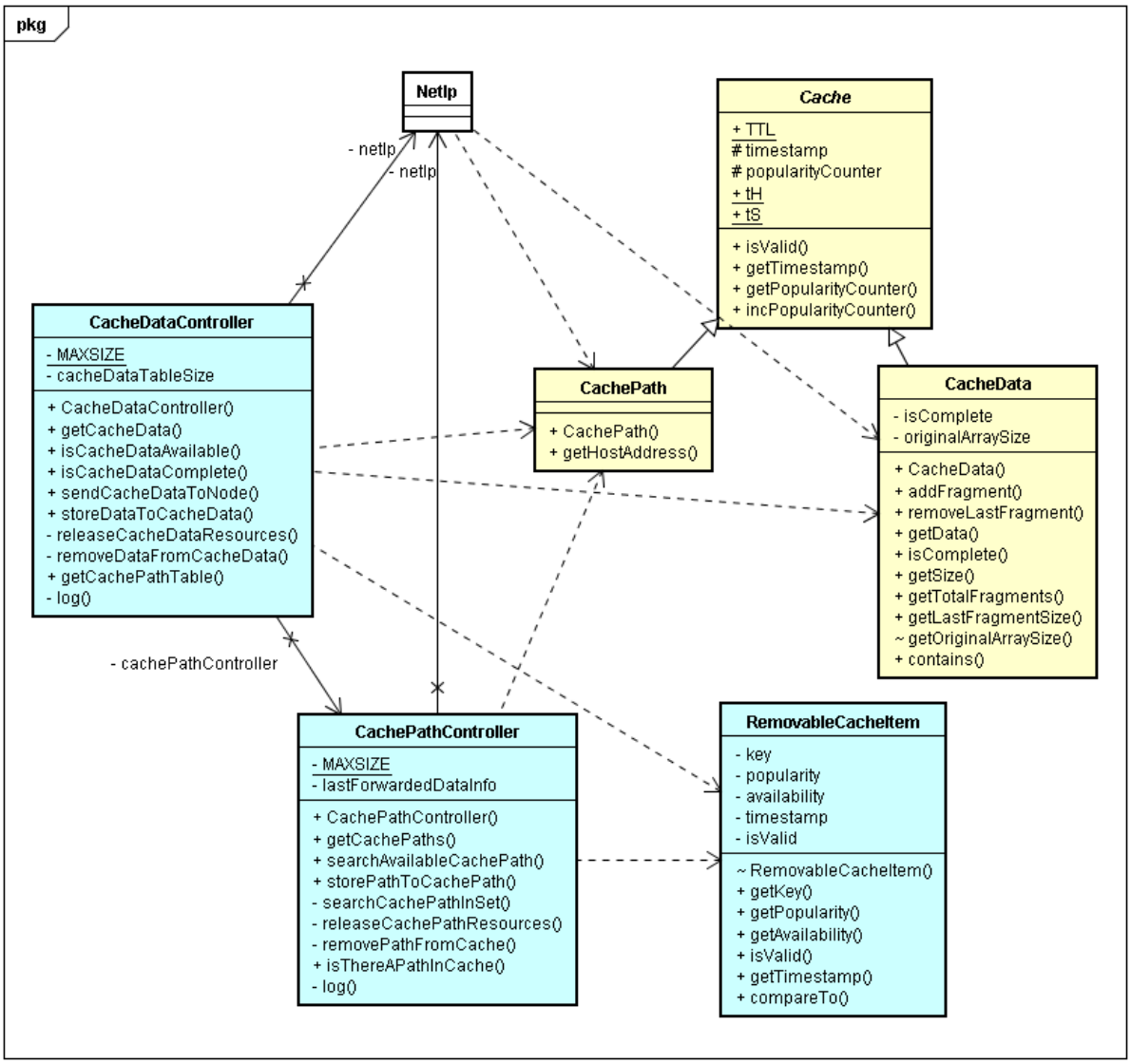

Figura 5.4: Arquitetura do tratamento de *cache* cooperativo

O protocolo de roteamento utilizado nas simulações foi o AODV [PER03], que possui implementação no SWANS. Para acessar a tabela de roteamento do protocolo AODV a partir da classe `jist.swans.net.NetIp`, alterações foram feitas na classe `jist.swans.route.RouteAodv`, que implementa o protocolo AODV, para a obtenção da sua tabela de rotas.

Para a simulação das mensagens GIOP, as classes *GIOPMessageRequest*, *GIOPMessageReply* e *GIOPMessageFragment* foram implementadas, que são subclasses da classe *GIOPMessage*. As classes *ServiceContext* e *ServiceContextList* foram implementadas

para simular as informações de contexto de serviço, contidas nas mensagens *Request* e *Reply* de CORBA. A Figura 5.5 ilustra o diagrama de relacionamento entre estas classes.

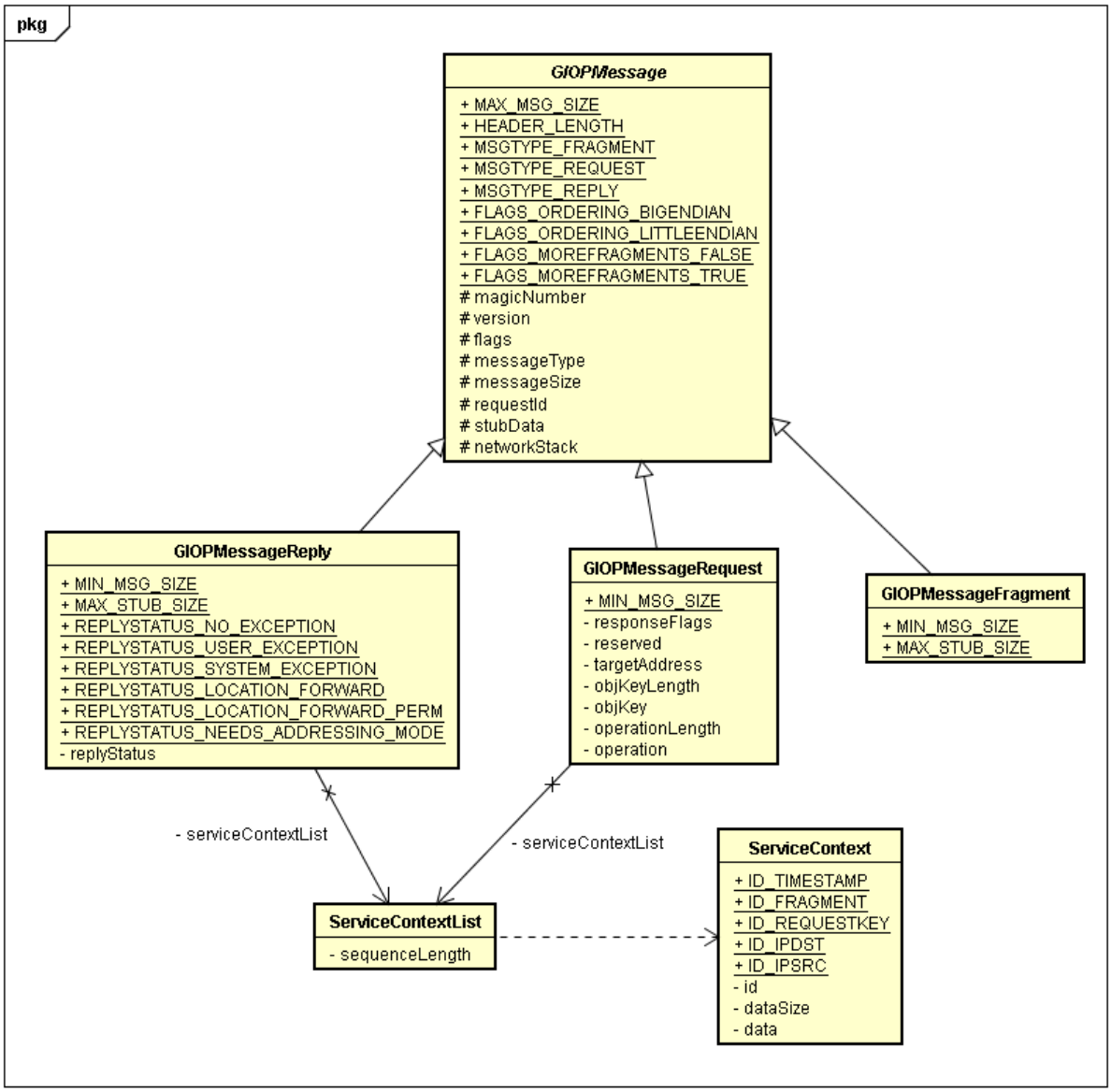

Figura 5.5: Classes de mapeamento das mensagens GIOP

Com relação às aplicações cliente e servidor que se comunicam nas simulações, foram implementadas as classes *GIOPClient*, *GIOPServer* e *GIOPServerThread*. *GIOPClient* implementa o ORB cliente que envia mensagens *Request* e espera pelas mensagens *Reply* e *Fragment*, provenientes do ORB servidor (*GIOPServer*), que cria uma instância de *GIOPServerThread* para cada requisição recebida.

A implementação das classes que definem as mensagens do protocolo GIOP foi feita com base na especificação CORBA [OMG04], além da análise de captura de tráfego de pacotes GIOP reais, entre dois computadores conectados como ORBs cliente e servidor. Esta captura de tráfego também serviu como base prática para a verificação exata de quais mensagens são trocadas entre os dois ORBs.

5.2.4. Ambientes de simulação

Para a avaliação do algoritmo de *cache* cooperativo, foram feitas simulações com base em dois ambientes diferentes. No primeiro ambiente, os nós se encontram em posições fixas e não possuem mobilidade, enquanto que no segundo, os nós se movimentam pela área estipulada. Os parâmetros de simulação baseados em [YIN04] comuns aos dois ambientes são descritos nesta seção. Com relação a cada ambiente, foram feitas duas simulações. A primeira compara o algoritmo proposto com um cenário sem tratamento de *cache* cooperativo, onde os dados de desempenho verificados estão relacionados a informações gerais, como pacotes enviados e recebidos pelo servidor de dados e tempo de resposta de nós clientes. A segunda simulação compara o algoritmo proposto com aquele apresentado em [YIN04], que faz o tratamento de *cache* cooperativo sem lidar com a fragmentação de *cache*, popularidade e disponibilidade. As comparações de desempenho foram feitas com relação ao espaço disponível de armazenamento em *cache* para cada nó.

Nas simulações, a banda de transmissão sem fio considerada foi de 1 Mb/s, para uma freqüência de transmissão de 2.4 GHz, com alcance de rádio de 400m. Os nós foram posicionados em pontos de uma área retangular definida para cada ambiente. Na área em questão, dois servidores de dados foram colocados em cantos opostos. Cinqüenta itens de dados foram colocados nos servidores, onde cada servidor mantém metade do total de dados. Estes dados não são alterados durante o tempo, ou seja, são estáticos. Itens de dados com identificadores pares foram salvos no servidor de endereço IP 0.0.0.202, enquanto que os itens de dados com identificadores ímpares foram salvos no servidor 0.0.0.101. O tamanho dos dados variou entre 256 e 12800 bytes, com o objetivo de ter, no mínimo, apenas mensagens GIOP *Reply* de resposta e, no máximo, uma mensagem GIOP *Reply* e treze mensagens GIOP *Fragment* subseqüentes de resposta. Cada mensagem contém, no máximo, 1KB de dados. Os servidores servem requisições em ordem de fila (FCFS - *first-come-first-served*). Com relação às consultas ao servidor enviadas pelos nós, cada consulta foi enviada somente depois que a consulta anterior tiver obtido uma resposta do servidor. Cada nó cliente

enviou trinta requisições de dados para os dois servidores. Quando a requisição era referente a um dado par, o servidor de endereço IP 0.0.0.202 era consultado. Caso contrário, o servidor de endereço IP 0.0.0.101 era consultado. Clientes de endereço IP com número equivalente à metade do número total de nós enviaram requisições referentes a itens de dados de 1 a 30, enquanto que o restante dos clientes enviou requisições de 20 a 50. Esta organização foi feita com o objetivo de alternar o fluxo de mensagens entre os nós da rede simulada, além de consultar os itens de número 20 a 30 de forma mais freqüente para a avaliação do tratamento de popularidade feito pelo algoritmo proposto. Para as simulações onde o tratamento de *cache* cooperativo é executado, seja ele o tratamento proposto ou o tratamento da Seção 3.1, foi considerado que todos os nós possuem tratamento de *cache* cooperativo, inclusive os nós servidores.

Para os tempos de resposta dos clientes, foram considerados somente os tempos de resposta das requisições bem sucedidas. Isso porque, em redes *ad hoc*, acontecimentos aleatórios estão implícitos ao envio de requisições pelos nós, como necessidade de retransmissão de pacotes não recebidos corretamente pelo cliente e descoberta de rota para os nós servidores. Estes casos elevam muito o tempo de resposta em algumas requisições, que acabam por afetar toda a simulação, onde a grande maioria dos tempos de resposta são menores.

Algumas considerações especiais foram feitas com relação ao envio de respostas a requisições. Isto porque, no simulador SWANS, para o envio de várias mensagens consecutivas referentes a uma resposta, um tempo mínimo de intervalo entre o envio de cada mensagem deve ser considerado. Caso contrário, os nós intermediários não recebem as mensagens, que acabam por não serem transmitidas para os nós clientes de destino. Foi então definida a seguinte fórmula para o cálculo do tempo de envio de cada resposta;

$$t_{envio} = 40 + (30 * \text{número de saltos até o destino}).$$

O valor de 40 milissegundos foi o valor mínimo de transmissão necessário constatado durante a execução das simulações, enquanto que o valor de 30 milissegundos acrescentado a cada salto da rota ao nó de destino serve para simular o acréscimo de tempo adicional por salto. Em redes *ad hoc* reais, cada salto consome mais tempo de transmissão e, nesta simulação, esta consideração serve para apresentar de forma mais consistente os resultados de

desempenho do algoritmo proposto, além do próprio método de *cache* cooperativo com relação ao cenário *ad hoc* sem tratamento de *cache*.

Outra consideração importante é a de que, para o tratamento de fragmentação de *cache* do algoritmo proposto, os nós servidores devem disponibilizar um grande espaço de armazenamento de memória *cache* das respostas enviadas, provavelmente maior do que a memória dos nós clientes. Caso contrário, muitos erros de *cache* (*cache misses*) podem acontecer, pois o servidor não terá o resto das mensagens GIOP *Fragment* solicitadas pelo nó cliente e irá retransmitir todas as mensagens que compõem a resposta a ser transmitida. Se o espaço de armazenamento não for consideravelmente alto, o número de erros de *cache* pode ser grande a ponto do algoritmo não ter nenhuma vantagem, podendo até ter resultados de desempenho inferiores a um modelo sem o tratamento de *cache*. Nas simulações, foi considerado que o espaço em armazenamento em *cache* disponível para os nós servidores era suficiente para armazenar todos os dados por ele disponibilizados, não havendo necessidade de exclusão destes itens de *cache*.

Para os resultados apresentados na Tabela 5.2 e Tabela 5.4, são apresentados alguns parâmetros de medição de desempenho, que são:

- **Acertos de *cache* local**: referente ao envio de respostas armazenadas em *cache* pelo próprio solicitante.
- **Acertos de *cache* remoto**: referente ao envio de respostas armazenadas em *cache* por nós intermediários na rota de transmissão da requisição ao nó servidor. Para o tratamento de fragmentação de *cache*, se um nó de *cache* enviou parte da resposta para o nó cliente e outro nó de *cache* enviou o restante da resposta, então o acerto de *cache* remoto é atribuído ao nó que enviou os dados de resposta finais.
- **Acertos de caminho de *cache* *(CachePath)***: referente ao envio de respostas armazenadas em *cache* por nós que receberam a requisição do cliente por meio de *CachePaths*, armazenados por outros nós de *cache*. O total destes acertos faz parte do total de acertos de *cache* remoto.
- **Erros de *cache* (*cache misses*)**: referente aos erros de *cache* que aconteceram na simulação. Os erros de *cache* são referentes a duas situações. Na primeira, o nó que recebeu a requisição por meio de um *CachePath* não possui mais a resposta em *cache* e encaminha a requisição ao servidor. Na segunda, que acontece por causa do tratamento de fragmentação de *cache*, o servidor não tem mais todas as mensagens

referentes à resposta solicitada pelo cliente armazenadas em *cache*, ou o TTL do dado armazenado em *cache* expirou, o que faz com que o servidor envie a resposta ao cliente como um tratamento normal de uma requisição. Ambos os casos acarretam em problemas de desempenho.

Com relação aos valores de *threshold* (seção 4.1.2) de ambas as simulações, T_S foi considerado como 5.222, que equivale a 40% da somatória do valor mínimo e máximo de resposta dos servidores, onde o valor mínimo é de 256 *bytes* e o valor máximo é de 12.800 *bytes*. Os outros parâmetros de *threshold* têm valores de configuração específicos, apresentados nas próximas seções.

5.2.5. Execução da simulação e resultados em uma rede *ad hoc* sem mobilidade

O ambiente de simulação considerado consiste em uma área retangular fixa de 4.000m x 6.000m. Os endereços IP estipulados dos nós clientes foram de 1 a 15 (0.0.0.1 a 0.0.0.15) no simulador. Clientes de endereço IP até o número 7 enviaram requisições referentes a itens de dados de 1 a 30, enquanto que os clientes de endereço IP de número superior a 8 enviaram requisições de 20 a 50.

Para os parâmetros de *threshold*, T_{TTL} foi considerado como 7.000 segundos, que equivale a metade do tempo total de simulação e T_H como 1, pois na rede deste ambiente não existem rotas muito longas para que um nó alcance um dos servidores. A Figura 5.6 apresenta o ambiente, enquanto que a Figura 5.7, a Figura 5.8 e a Tabela 5.1 mostram os dados obtidos da simulação do primeiro cenário, que compara a execução do algoritmo com o cenário de rede *ad hoc* sem qualquer tratamento de *cache*.

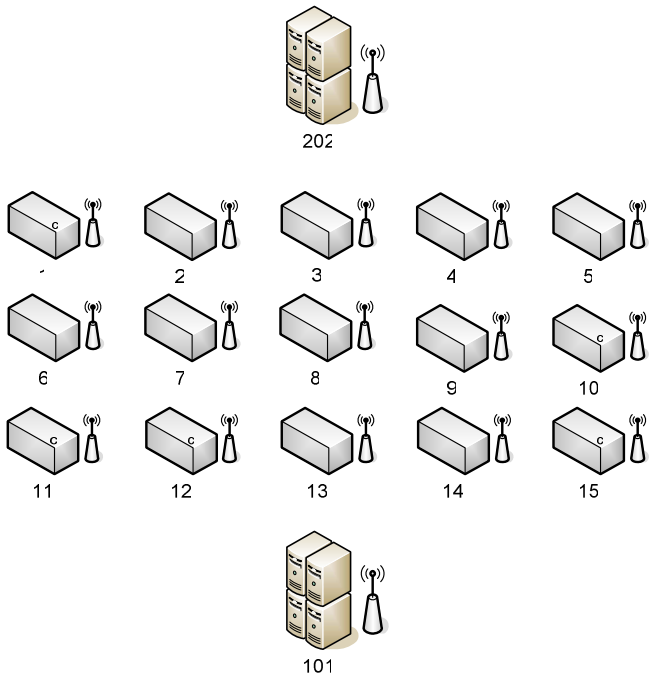

Figura 5.6: Ambiente de simulação de uma rede *ad hoc* com nós sem mobilidade.

	Sem tratamento	**Algoritmo proposto**	**Ganho (%)**
Mensagens recebidas pelo servidor	454	153	66,3
Mensagens enviadas pelo servidor	3379	1249	63
Mensagens enviadas por acerto de *cache*	0	2128	-
Tempo de resposta de nós clientes (ms)	776	555	28,47

Tabela 5.1: Dados de desempenho do algoritmo de *cache* cooperativo proposto com relação ao cenário sem tratamento de *cache*, em uma rede *ad hoc* sem mobilidade.

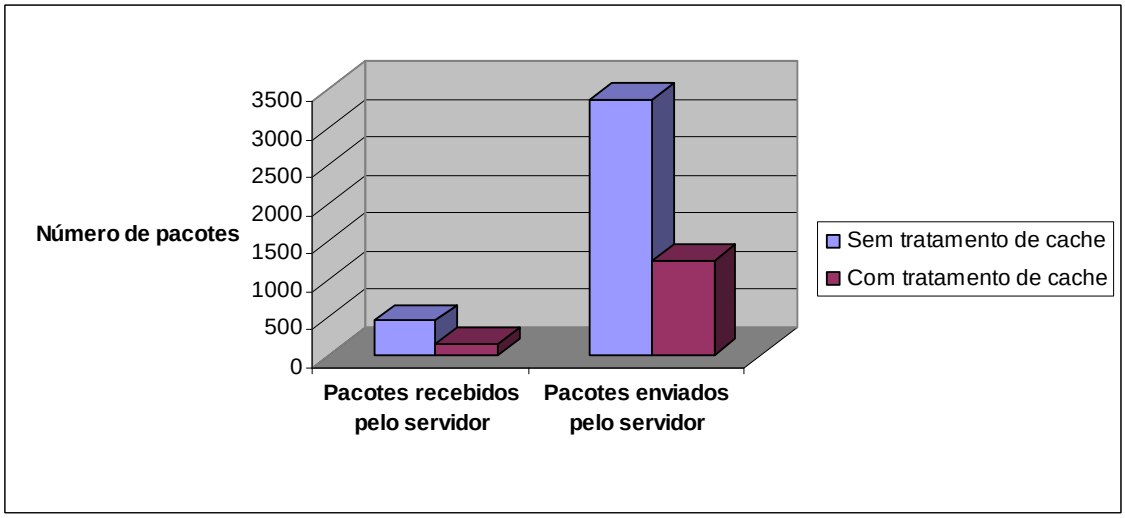

Figura 5.7: Comparação de dados de pacotes enviados e recebidos ao servidor, em uma rede *ad hoc* sem mobilidade.

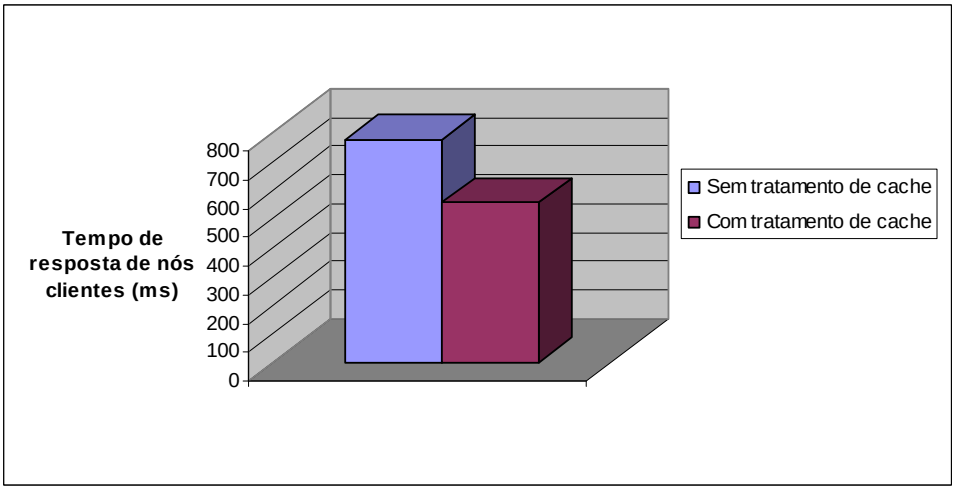

Figura 5.8: Comparação de tempos de resposta dos nós clientes, em uma rede *ad hoc* sem mobilidade.

Como pode ser visto na Tabela 5.1, Figura 5.7 e Figura 5.8, houve um ganho considerável de desempenho com relação ao tempo de espera dos clientes pelas respostas a suas requisições. O ganho médio de desempenho foi de 28,47%. Também pode ser observado que o número de mensagens enviadas e recebidas pelo servidor teve uma redução de 66,3%, onde a maior parte das mensagens foi enviada pelos próprios nós de *cache*. Além de melhorar o desempenho de processamento do servidor, isso contribui para melhores tempos de resposta, que não foram considerados nesta simulação.

Com relação à segunda simulação, que compara o desempenho do algoritmo proposto com o algoritmo de [YIN04], a Figura 5.9, a Figura 5.10 e a Tabela 5.2 apresentam os dados obtidos, onde a Figura 5.9 apresenta a informação sobre os acertos de *cache* como a soma das informações sobre acerto de *cache* local, remoto e de caminho de *cache*, presentes na Tabela 5.2. Nesta simulação, os dados armazenados em *cache* foram variados para se observar os efeitos do tratamento de fragmentação de *cache* no comportamento do algoritmo, além dos tratamentos de popularidade e disponibilidade.

Memória disponível em *cache* (KB)	**512**		**256**		**128**		**64**		**32**	
Algoritmo	**Orig.**	**Prop.**	**Orig.**	**Prop.**	**Orig.**	**Prop.**	**Orig.**	**Prop.**	**Orig.**	**Prop.**
Acertos de *cache* local	64	65	65	66	63	65	66	56	17	19
Acertos de *cache* remoto	209	234	204	231	168	202	119	151	30	34
Acertos de caminho de *cache* (*CachePath*)	1	0	1	0	2	0	2	0	0	0
Erros de *cache* (*cache miss*)	0	0	0	0	0	0	0	2	1	2
Tempo de resposta de nós clientes (ms)	555	555	559	561	626	600	683	668	752	753

Tabela 5.2: Dados de desempenho do algoritmo proposto com relação ao algoritmo de *cache* cooperativo de [YIN04], apresentado na Seção 3.1, em uma rede *ad hoc* sem mobilidade.

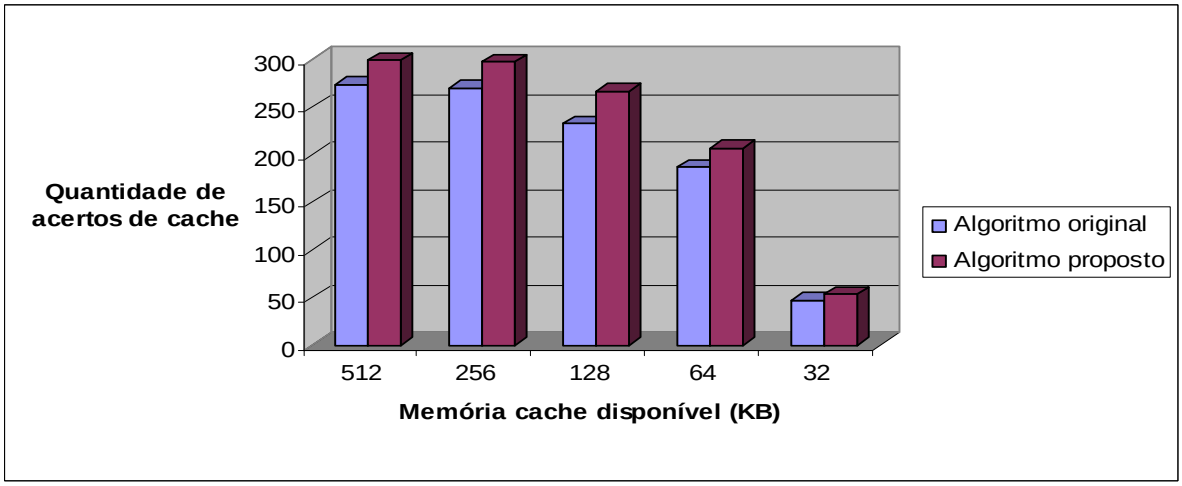

Figura 5.9: Comparação da quantidade de acertos de *cache* pela memória de *cache* disponível, em uma rede *ad hoc* sem mobilidade.

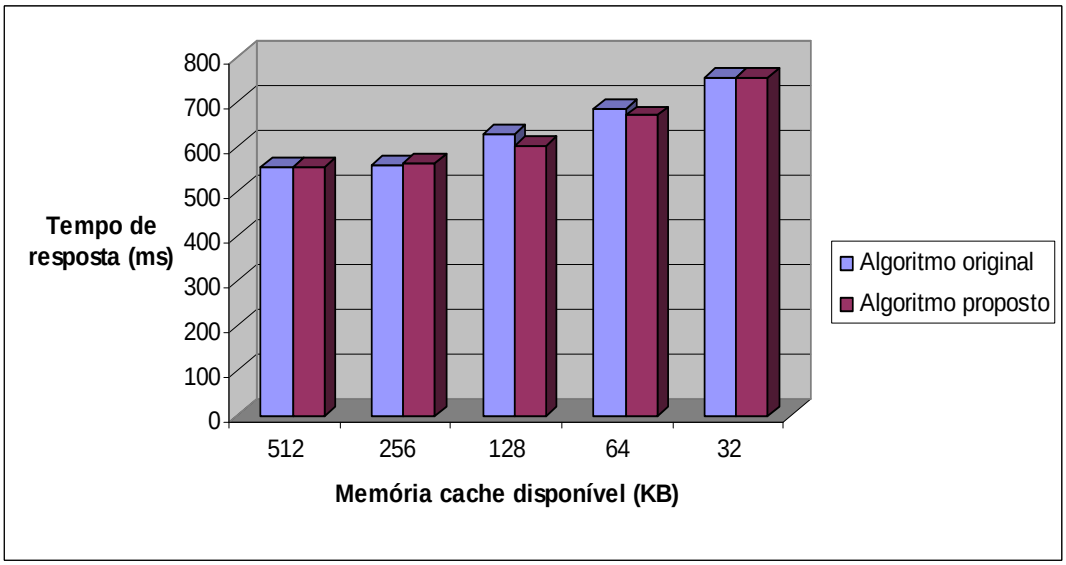

Figura 5.10: Comparação do tempo de resposta dos nós com relação à memória de *cache* disponível, em uma rede *ad hoc* sem mobilidade

.

Os dados da Tabela 5.2 referentes às quantidades de memória em *cache* de 512 e 256 KB não apresentaram melhoras com relação ao algoritmo proposto. Isto porque as suas vantagens só podem ser avaliadas no momento em que a exclusão de dados em *cache* se faz necessária, e isso só começa a acontecer a partir do ponto em que há 256 KB disponíveis em *cache*. Para 512 KB disponíveis, não há exclusão de itens armazenados. Todavia, as melhoras de acerto de *cache* remoto são constatadas em todos os casos, o que faz com que esse método melhore o tempo de resposta dos servidores. Isto aconteceu devido à prioridade de armazenamento de itens de *cache* mais relevantes, pelo tratamento de popularidade e disponibilidade, aumentando os acertos de *cache* local e remoto. Por este mesmo motivo, o número de acertos de caminho de *cache* diminuiu, pois os nós de *cache* que enviariam a requisição para outro nó ainda tinham a resposta requerida armazenada em *cache*. Os casos que tiveram o melhor desempenho de execução do algoritmo foram para as quantidades de 128 KB e de 64 KB disponíveis em *cache*, chegando a cerca de 4,15% de ganho de desempenho para a quantidade de 128 KB. A diferença de desempenho entre os dois métodos foi equivalente para 32 KB de memória *cache* disponível, pois o número de itens de *cache* armazenado foi baixo. Logo, a melhoria de desempenho para o método de fragmentação de *cache* foi constatada para os casos onde os índices de acerto de *cache* local e remoto foram mais significativos.

5.2.6. Execução da simulação e resultados em uma rede *ad hoc* móvel

Para a simulação de uma rede *ad hoc* móvel, o ambiente de simulação considerado consiste em uma área retangular de 2.400m x 512m, que são dimensões proporcionais às dimensões do ambiente apresentado em [YIN04], em função do alcance de rádio considerado nesta simulação, de 400m. Os endereços IP estipulados dos nós clientes foram de 1 a 100 (0.0.0.1 a 0.0.0.100) no simulador, onde apenas 25 destes nós enviam requisições. Clientes de endereço IP até o número 50 enviaram requisições referentes a itens de dados de 1 a 30, enquanto que os clientes de endereço IP de número superior a 51 enviaram requisições de 20 a 50. Os cem nós foram posicionados em pontos aleatórios da área de simulação e se movimentaram segundo o Modelo de Mobilidade de Ponto de Mudança de Rota (*Random Waypoint Mobility Model*) [PER04]. Depois de se movimentar em uma velocidade de 4 metros por segundo, cada nó pára por 30 segundos antes de repetir o seu movimento aleatório.

Para os parâmetros de *threshold*, T_{TTL} foi considerado como 125.000 segundos, que equivale a metade do tempo total de simulação e T_H como 1. O tempo de simulação teve uma duração maior para evitar interferências entre nós que tiveram demoras na obtenção das respostas do servidor. De acordo com o ambiente apresentado em [YIN04], o fator T_H utilizado foi 2 mas, devido ao baixo número de caminhos de *cache* armazenados, o valor foi considerado como 1. A Figura 5.11 apresenta uma ilustração do ambiente, enquanto que a Figura 5.12, a Figura 5.13 e a Tabela 5.3 mostram os dados obtidos da simulação do primeiro cenário, que compara a execução do algoritmo com o cenário de rede *ad hoc* sem qualquer tratamento de *cache*.

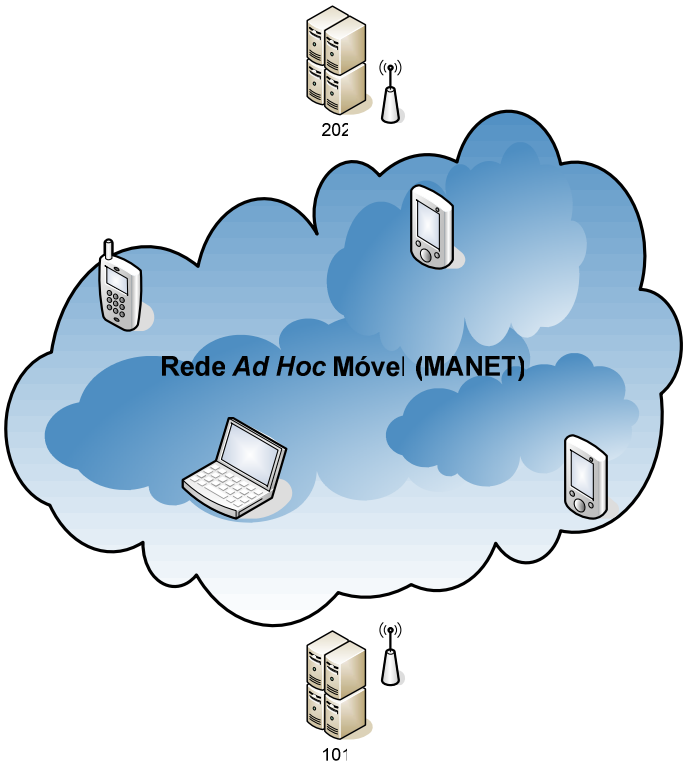

Figura 5.11: Ambiente de simulação de uma rede *ad hoc* móvel.

	Sem tratamento	**Algoritmo proposto**	**Ganho (%)**
Mensagens recebidas pelo servidor	899	783	12,9
Mensagens enviadas pelo servidor	6391	5619	12,08
Mensagens enviadas por acerto de *cache*	0	682	-
Tempo de resposta de nós clientes (ms)	502	475	5,37

Tabela 5.3: Dados de desempenho do algoritmo de *cache* cooperativo proposto com relação ao cenário sem tratamento de *cache*, em uma rede *ad hoc* móvel

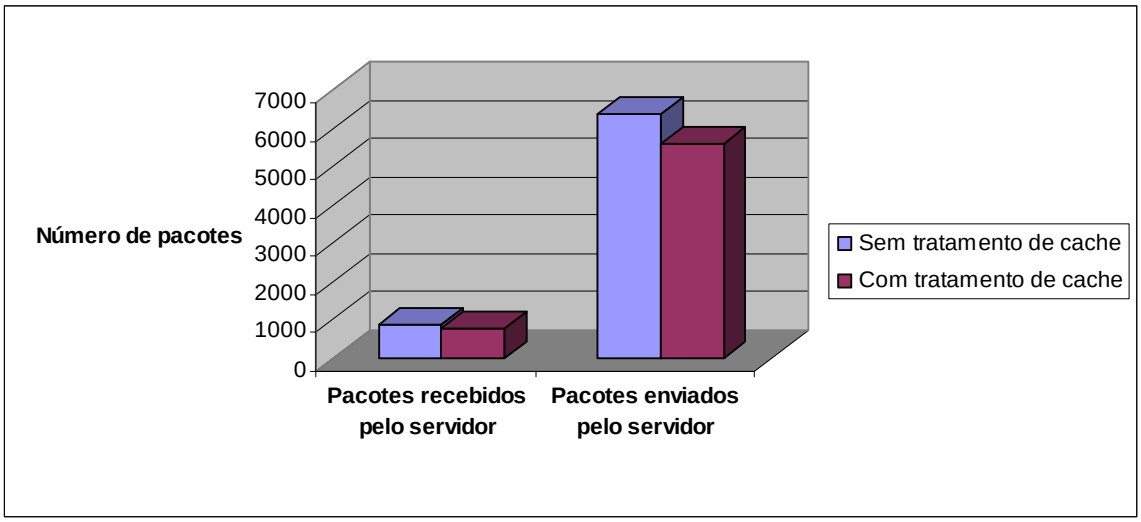

Figura 5.12: Comparação de dados de pacotes enviados e recebidos ao servidor, em uma rede *ad hoc* móvel

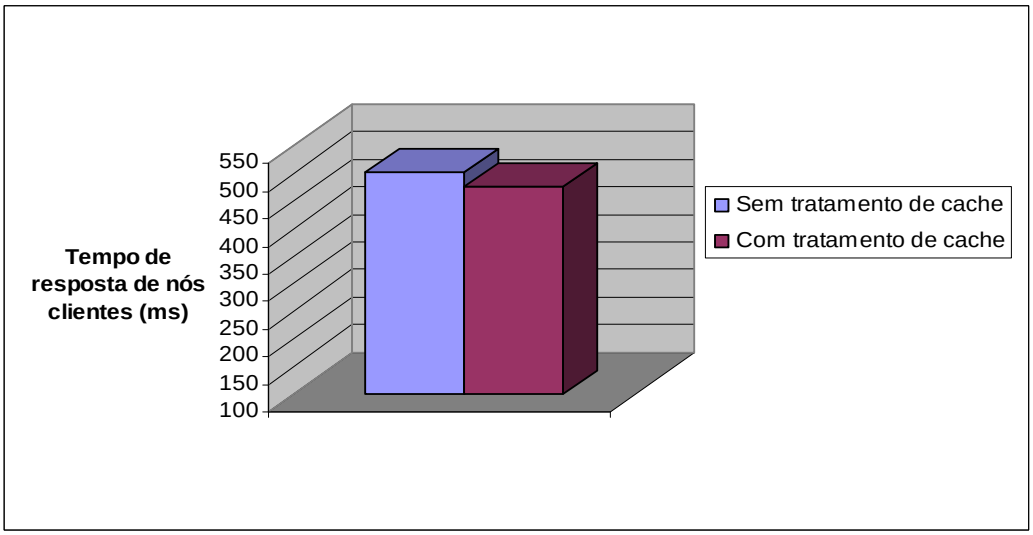

Figura 5.13: Comparação de tempos de resposta dos nós clientes, em uma rede *ad hoc* móvel.

Devido ao cenário móvel da rede *ad hoc*, o desempenho da aplicação do algoritmo não é alto como no cenário sem mobilidade. Como pode ser visto na Tabela 5.3, o ganho de desempenho foi de 5,37%. Para um valor de TTL com a duração da própria simulação, onde o TTL dos dados em *cache* não expira, o tempo de resposta médio dos nós cliente ficou em 460 ms, obtendo-se 8,36% de ganho de desempenho. Quanto ao número de mensagens enviadas e recebidas pelo servidor, os ganhos de desempenho foram de 12,08% e 12,9%, respectivamente.

Com relação à simulação que compara o desempenho do algoritmo proposto com o algoritmo de [YIN04], a Figura 5.14, a Figura 5.15 e a Tabela 5.4 apresentam os dados

obtidos. O propósito desta simulação foi o mesmo da simulação realizada no cenário de redes *ad hoc* sem mobilidade, com o intuito de avaliar o desempenho do algoritmo proposto no contexto de redes *ad hoc* móveis, variando-se o tamanho da memória de armazenamento em *cache*.

Memória disponível em *cache* (KB)	**512**		**256**		**128**		**64**		**32**	
Algoritmo	**Orig.**	**Prop.**	**Orig.**	**Prop.**	**Orig.**	**Prop.**	**Orig.**	**Prop.**	**Orig.**	**Prop.**
Acertos de *cache* local	28	20	32	20	29	23	44	37	23	25
Acertos de *cache* remoto	34	53	22	53	21	36	35	40	9	48
Acertos de caminho de *cache* (*CachePath*)	0	0	0	0	0	0	0	0	0	0
Erros de *cache* (*cache miss*)	0	14	0	14	0	15	0	14	0	8
Tempo de resposta de nós clientes (ms)	472	475	475	475	483	465	465	460	475	486

Tabela 5.4: Dados de desempenho do algoritmo proposto com relação ao algoritmo de *cache* cooperativo de [YIN04], apresentado na Seção 3.1, em um rede *ad hoc* móvel.

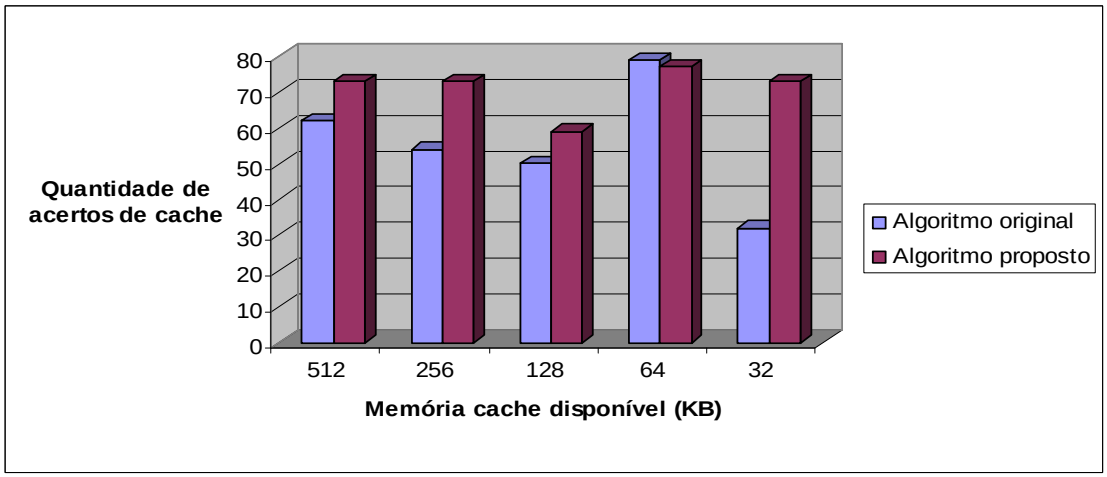

Figura 5.14: Comparação da quantidade de acertos de *cache* pela memória de *cache* disponível, em uma rede *ad hoc* móvel.

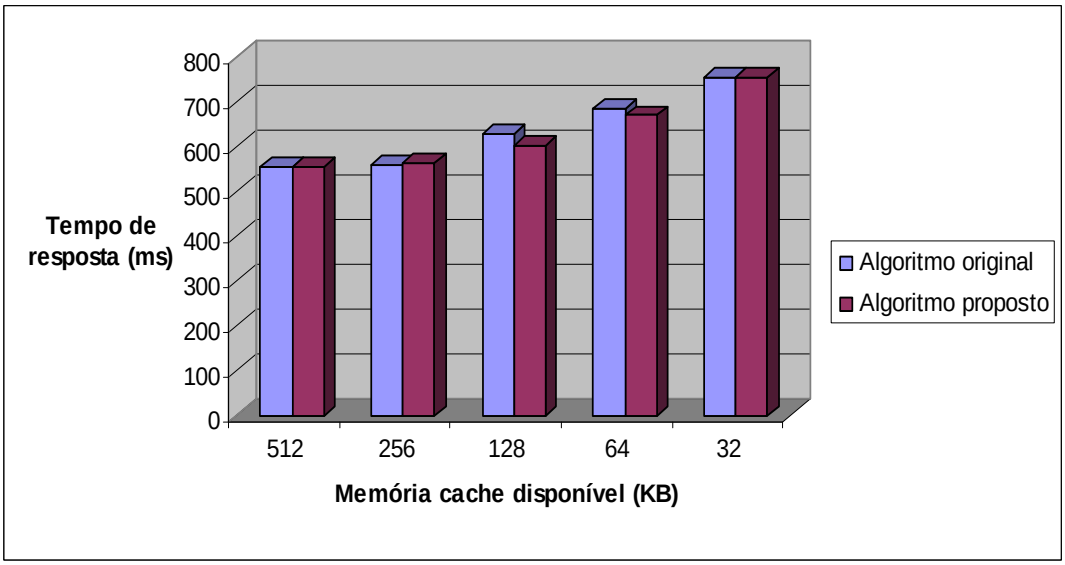

Figura 5.15: Comparação do tempo de resposta dos nós com relação à memória de *cache* disponível, em uma rede *ad hoc* sem mobilidade

Assim como na simulação anterior, a mobilidade dos nós afetou os resultados da simulação. Em cenários em que os nós têm menos espaço de armazenamento em *cache* do que outros cenários obtiveram-se tempos de resposta melhores para os nós clientes, o que é uma inconsistência. Isto aconteceu pelo fato de, em algumas situações, os nós clientes estarem mais próximos dos servidores, ou devido à mobilidade de algum nó intermediário, que gerou a necessidade de redescoberta da rota de comunicação com o servidor. Como estes acontecimentos são aleatórios, estas situações podem acontecer com mais freqüência em alguns cenários do que em outros, gerando as inconsistências de desempenho observadas nos resultados das simulações de redes *ad hoc* móveis, principalmente do contexto de operação apresentado, onde as respostas às requisições são fragmentadas em várias mensagens.

Feitas estas considerações, os resultados de desempenho desta simulação foram analisados somente de forma comparativa entre os resultados referentes à mesma quantidade de memória *cache* disponível. Pode-se observar que, para 512 KB e 256 KB, os dois algoritmos são equivalentes, onde o algoritmo de [YIN04] tem um desempenho um pouco melhor para 512 KB. Para 128 KB e 64 KB, onde há um maior descarte de dados em *cache*, o algoritmo proposto tem um desempenho melhor com relação ao outro algoritmo, com ganhos de 3,72% para 128 KB e 1% para 64 KB. Para 32 KB, o algoritmo proposto tem um desempenho inferior ao algoritmo de [YIN04]. Os resultados desta simulação foram similares à simulação feita para redes *ad hoc* sem mobilidade, mas o ganho de desempenho do algoritmo proposto foi inferior ao contexto sem mobilidade. Pode-se observar também que

houve um grande aumento de erros de *cache* no ambiente móvel. Isso aconteceu porque o TTL dos dados em *cache* mantidos no nó servidor expirou e, nas simulações de rede *ad hoc* móvel, as requisições a serem atendidas por fragmentos de *cache* inicial foram mais freqüentes, gerando os erros de *cache* constatados.

5.3. Conclusão

Este capítulo apresentou o modelo de implementação do algoritmo de *cache* cooperativo proposto neste trabalho no contexto de MANETs. A implementação e os resultados da execução da simulação foram descritos, e detalhes sobre os resultados da simulação foram apresentados.

Durante a execução da simulação, foi constatado que o uso excessivo de *CachePath* resulta em muitos problemas, caso o *CachePath* não seja mais válido. *CachePath* indica a localização de um nó de *cache* próximo, que contém o *CacheData* requerido pelo nó cliente. Em redes dinâmicas como MANETs, estas rotas podem se tornar obsoletas facilmente, e é por isso que o algoritmo procura minimizar a utilização do *CachePath* utilizando o fator H_{SAVE} para tanto, descrito no Capítulo 4. A constatação das conseqüências negativas do uso excessivo do *CachePath* ocorreu durante a implementação do algoritmo descrito em [DAS04], que é uma variação do algoritmo descrito na Seção 3.1 que desconsidera o fator H_{SAVE} durante o armazenamento de *CachePaths*. Além disso, a exclusão de dados em *cache* por cada nó de *cache* faz com que um dado que antes estava disponível não possa mais ser obtido no futuro. O *CachePath* poderia ser melhor utilizado se houvesse um protocolo de comunicação entre os nós de *cache*, para que estes nós se comunicassem antes de enviar uma requisição de um dado. Isso evitaria a possibilidade de erros de *cache*, inclusive no tratamento de fragmentação de *cache* apresentado nesta dissertação.

Nos ambientes de simulação implementados, foi verificado que *CachePaths* são armazenados poucas vezes. Estes ambientes foram implementados com base no modelo de simulação descrito em [YIN04], que descreve um ambiente relativamente pequeno para que se formem rotas extensas para *CachePaths*. Logo, conclui-se que a freqüência de armazenamento de *CachePath*s depende muito das dimensões da rede *ad hoc*. A dificuldade de se analisar os efeitos de *CachePaths* neste trabalho impossibilitou também a verificação efetiva do método de disponibilidade na remoção de itens de *cache*, pois este tratamento está

diretamente relacionado com a quantidade de *CachePaths* associados a um item de *CacheData*.

Foi constatado também que, pelo tratamento de *cache* disponibilizar dados de forma distribuída na rede, as requisições dos clientes são atendidas mais rapidamente também pelo fato de que os nós clientes muitas vezes não conseguirem se comunicar como o nó que contém o objeto com o método requerido, pelo fato do servidor estar distante ou de, simplesmente, o protocolo de roteamento não conseguir encontrar uma rota para o servidor. Logo, nós que tenham a resposta solicitada pelo cliente em *cache* que possam simplesmente ser alcançados contribuem para um melhor desempenho geral de transmissão de dados na rede.

Conclusão

Este trabalho apresentou um algoritmo de *cache* cooperativo aplicado ao protocolo GIOP, utilizado para a comunicação entre ORBs com o objetivo de otimizar a comunicação entre nós em uma rede não estruturada. Após a definição de dois cenários de implementação do algoritmo (discutidos na Seção 2.10) e do estudo comparativo entre eles, a implementação do algoritmo em uma rede móvel foi discutida no Capítulo 5, bem como os resultados da simulação executada.

Muito ainda deve ser feito em redes *ad hoc* móveis com relação à padronização de protocolos e interoperabilidade de aplicações provenientes de redes fixas com MANETs. A maior integração entre a camada de aplicação e a camada de rede do modelo OSI é uma das questões chave para a melhoria de desempenho de transmissão de dados entre dispositivos atuando nestas redes. Os estudos recentes da técnica de *cache* cooperativo em MANETs visam tratar o problema de desempenho de transmissão, propondo técnicas que armazenamento de *cache* de forma distribuída entre os dispositivos de redes deste tipo. A implementação destas técnicas está fortemente relacionada com a integração entre as camadas de aplicação e de rede do modelo OSI, e foi esta linha de implementação que também foi abordada neste trabalho, inclusive nas simulações. Apesar da execução das simulações ter sido feita diretamente na camada de rede utilizando-se o protocolo UDP, onde a justificativa da validade dos dados de simulação encontra-se na Seção 5.2.1, pode-se constatar uma grande diferença de desempenho, comparando-se o cenário de redes *ad hoc* móveis sem e com a aplicação do algoritmo. Foi constatada também uma melhora de desempenho significativa com a aplicação da técnica de fragmentação de *cache*, contribuição científica apresentada na Seção 4.2.6. Este trabalho pode servir como uma base inicial de estudos sobre *cache* cooperativo aplicado ao protocolo GIOP e como um complemento à pesquisa sobre a adaptação da tecnologia CORBA ao contexto de MANETs. Várias considerações devem ser feitas para fazer com que CORBA possa ser utilizada em redes deste tipo, e espera-se que esta dissertação possa ajudar nesta tarefa.

Quanto a trabalhos futuros, muitos pontos a serem trabalhados em CORBA com relação a MANETs foram identificados. Embora este trabalho tenha estudado o desempenho do algoritmo por meio de simulações, uma implementação real do algoritmo em, pelo menos, dois ou três computadores para verificar questões de implementação reais poderá ser feita. Vários estudos sobre consistência de *cache* poderiam ser aplicados no contexto deste trabalho, verificando possíveis aplicações e adaptações destas técnicas [CAO05][COO06]. Ao desenvolver o algoritmo deste trabalho, foi constatado também que o estudo de um protocolo de comunicação pode ser formulado para atuar em conjunto com o algoritmo, provendo comunicação entre os módulos de *cache* para que estes possam trocar informações sobre quais itens de *cache* cada módulo tem, etc. Conceitos como *cache* local e global, apresentados na Seção 3.2, podem ser viabilizados com a utilização de um protocolo como este. Existem também alguns trabalhos de tratamento de *cache* em CORBA, voltados em sua maioria para o tratamento de *cache* de objetos CORBA no contexto de redes cabeadas, sendo que alguns deles foram identificados com relação a *cache* cooperativo para redes cabeadas e *cache* não cooperativo para redes móveis, como apresentado na Seção 3.5. Um estudo comparativo mais aprofundado entre estes trabalhos e o contexto de redes *ad hoc* poderá ser feito, comparando-se a técnica de *cache* de objetos CORBA com a técnica apresentada nesta dissertação, relacionada a *cache* de mensagens *Reply* e *Fragment* CORBA. Como já comentado na Seção 2.10, um protocolo de roteamento integrado com ORBs da camada de aplicação poderá descobrir rotas aos nós da rede e armazenar quais objetos CORBA estão presentes nestes nós, melhorando ainda mais o desempenho de procura de objetos CORBA efetuado pela aplicação do algoritmo BFS à arquitetura *Ad Hoc* CORBA (Seção 2.9.2). Pesquisas sobre a aplicação da especificação CORBA de MIOP (*Unreliable Multicast Inter-ORB Protocol*)[OMG01][BES02] também seriam úteis neste contexto. Além disso, o conceito tratado neste trabalho sobre *cache* cooperativo poderia também ser aplicado em redes cabeadas, onde estudos sobre a aplicação deste algoritmo neste contexto poderiam ser realizados, além de uma comparação com as técnicas já existentes de tratamento de *cache* para este tipo de redes. Quanto à compressão de dados, um protocolo denominado ZIOP (GIOP *Compression*) está em processo de integração à especificação CORBA [CAC08]. Trabalhos futuros podem integrar este protocolo ao método de *cache* cooperativo em MANETs, com o objetivo de melhorar ainda mais o desempenho de transmissões GIOP nesse contexto.

Referências Bibliográficas

[AUN04] AUNE, F. *Cross-Layer Design Tutorial*. Norwegian University of Science and Technology, Dept. of Electronics and Telecommunications 2004.

[BAR04] BARR, R. *JiST – Java in Simulation Time User Guide*. http://jist.ece.cornell.edu/docs/040319-jist-user.pdf. Cornell Research Foundation, 2004. Acesso em 26/01/2009.

[BES02] BESSANI, A.; FRAGA, J. S; LUNG, L. C. *MJaco – Integração do Multicast IP na Arquitetura CORBA*. Laboratório de Controle e Microinformática (LCMI) – DAS – CTC – UFSC.

[BOS05] BÖSE, J; BÖTTCHER, S; GRUENWALD, L; MARRÓN, P; OBREITER, P; PITOURA, E; REIHER, P; SATTLER, K; SELIGER, F. *Some Open Aspects of Mobile Ad-hoc NETwork, Peer-to-Peer, and Self-organizing Systems*. Dagstuhl Seminar Proceedings, http://drops.dagstuhl.de/opus/volltexte/2005/217, 2005, Acesso em 27/03/2007.

[CAC08] CACERES, J.; WILLEMSEN, J. *GIOP Compression RFP initial submission*. https://svn.dre.vanderbilt.edu/viewvc/Middleware/trunk/TAO/docs/ZIOP.pdf?revision=82358. OMG, May, 2008. Acesso em 16/02/2009.

[CAO05] CAO, J; ZHANG, Y; XIE, L; CAO, G. *Consistency of cooperative caching in mobile peer-to-peer systems over MANET*. IEEE ICDCSW'05, 2005, p.573-579.

[CHO00] CHOCKLER, G; FRIEDMAN, R; VITENBERG, R. *Consistency Conditions for a CORBA Caching Service*. Lecture Notes In Computer Science; Vol. 1914 archive,

Proceedings of the 14th International Conference on Distributed Computing table of contents, pp. 374-388, 2000.

[COO06] CAO, J. *Cooperative Caching in Mobile P2P Systems over MANET*. SBRC 2006, 2006.

[DAS04] DAS, C. R.; CAO, G.; YIN, L. *Cooperative Cache-Based Data Access in Ad Hoc Networks*. IEEE, Vol 37, Issue 2, p. 32-39, Feb, 2004.

[DEN05] DENNING, P. J. *The Locality Principle*. Communications of the ACM, 2005, Vol. 48, No. 7.

[DOD04] DODERO, G; GIANUZZI, V; MURA, A; PASTORINO, L. *MobEYE, a Mobile intercEpting proxY cachE for MANET*. DISI, Universita' di Genova, 2004.

[ERI94] ERIKSSON, H. *MBONE: the multicast backbone*. Communications of the ACM, 37(8):54–60, 1994.

[FEE99] FEENEY, L. M. *A Taxonomy for Routing Protocols in Mobile Ad hoc Networks*. SICS Technical Report T99/07, October, 1999.

[FRO00] FRODIGH, M; JOHANSSON, P; LARSSON, P. *Wireless ad hoc networking - The art of networking without a network*. Ericsson Review No. 4, 2000.

[GEN04] GENUA, P. *A Cache Primer*. Freescale Semiconductor, Inc., 2004.

[GER05] GERLA, M; LINDEMANN, C; ROWSTRON, A. *P2P MANETs – New Research Issues*. Dagstuhl Seminar Proceedings, http://drops.dagstuhl.de/opus/volltexte/2005/213, 2005. Acesso em 26/02/2007.

[GOK02] GOKHALE, A; KUMAR, B.; SAHUGUET A. *Reinventing the Wheel? CORBA vs. Web Services*. WWW2002 Conference Proceedings, 2002.

[HAM01] HAMIDJAJA, H.; TARI, Z. *A CORBA Cooperative Cache Approach with Popularity Admision and Routing Mechanism*. Thirteenth Australasian Database Conference (ADC2002), Melbourne, Australia, Conferences in Research and Practice in Information Technology, Vol. 5., 2001.

[HEN06] HENNING, M. *The Rise and Fall of CORBA*. ACM Queue, 2006.

[JAY04] JAYAPUTERA, J.; TANIAR, D. *Invalidation for CORBA Caching in Wireless Devices*. L.T. Yang et al. (Eds.): EUC 2004, LNCS 3207, pp. 460–471, 2004.

[JIS04] JiST / SWANS - Java in Simulation Time / Scalable Wireless *Ad hoc* Network Simulator. http://jist.ece.cornell.edu/. Cornell Research Foundation, 2004. Acesso em 10/05/2007.

[LIM07] LIMA, L.; CALSAVARA, A. *A Framework for CORBA Interoperability in Ad hoc Networks*. Proceedings of the 2007 ACM SIGAPP, 2007.

[MCH07] McHALE, C. *CORBA Explained Simply*. http://www.ciaranmchale.com/corba-explained-simply/, 2007. Acesso em 26/02/2009.

[OMG01] OMG. *Unreliable Multicast Inter-ORB Protocol Specification*. 2001, version 1.0.

[OMG03] OMG. *Wireless Access and Terminal Mobility in CORBA*. 2003, version 1.0, formal/03-03-64.

[OMG04] OMG. *Common Object Request Broker Architecture: Core Specification*. March 2004, version 3.0.3, formal/04-03-12.

[OMG08] OMG. *CORBA To WSDL/SOAP Interworking (C2WSDL)*. 2008, version 1.2.1, formal/2008-08-03.

[PER03] PERKINS, C.; BELDING-ROYER, E.; DAS, S. *Ad hoc On-Demand Distance Vector (AODV) Routing*. http://www.ietf.org/rfc/rfc3561.txt. Nokia Research Center; University of California, Santa Barbara; University of Cincinnati, July 2003.

[PER04] PEREIRA, I. C. *Análise do roteamento em redes móveis ad hoc em cenários de operações militares*. Dissertação de Mestrado, Universidade Federal do Rio de Janeiro, 2004.

[POL08] POLO, R.; LIMA, L. *Cache Cooperativo Aplicado ao Protocolo GIOP em MANETs*. LTPD, 2008.

[SQU09] *Squid: Optimising Web Delivery*. http://www.squid-cache.org/. Acesso em 27/01/2009.

[SWA07] *SWANS++ - Extensions to the Scalable Wireless Ad-hoc Network Simulator*. http://www.aqualab.cs.northwestern.edu/projects/swans++/. AquaLab - Research in Distributed Computing, 2007. Acesso em 15/10/2007.

[SUN02] Sun Microsystems. *Java RMI over IIOP Technology*. http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/. 2002.

[SUN03] SunONE. *Configuring the Server For Corba/IIOP Clients*. http://docsun.cites.uiuc.edu/sun_docs/C/solaris_9/SUNWadoc/SONEAPPSVRADMIN/agcorba.html. Sun ONE Application Server 7, Update 1 Administrator's Guide, 2003. Acesso em 16/02/2009.

[TAN05] TANG, B; ZHOU, Z; KASHYAP, A; CHIUEH, T. *An integrated approach for P2P file sharing on multi-hop wireless networks*. IEEE WiMob'2005, Vol.3, 2005, p.268-274.

[TAR01] TARI, Z; BUKHRES, O. *Fundamentals of Distributed Object Systems: The CORBA Perspective - CORBA Caching*. Wiley Series on Parallel and Distributed Computing, pp. 183-210, 2001.

[VIN01] VINOSKI, S; SCHMIDT, D. *Object Interconnections: Real-time CORBA, Part 1: Motivation and Overview*. TechWeb, http://www.ddj.com/cpp/184403809, 2001. Acesso em 25/02/2009.

[WES01] WESSELS, D. *Web Caching*. O'Reilly, 2001.

[WES04] WESSELS, D. *Squid: The Definitive Guide*. O'Reilly, 2004.

[WU06] WU, W; TAN, K. *Global Cache Management in Nonuniform Mobile Broadcast*. IEEE Proceedings of the 7th International Conference on Mobile Data Management (MDM'06), 2006.

[YIN04] YIN, L.; CAO, G. *Supporting Cooperative Caching in Ad hoc Networks*. IEEE INFOCOM 2004.

[ZHA08] ZHAO, J.; ZHANG, P; CAO, G. *On Cooperative Caching in Wireless P2P Networks*. Department of Computer Science and Engineering, Pennsylvania State University, 2008.