Universidade Federal de Pernambuco

Engenharia da Computação

Centro de Informática

Conectores baseados em RabbitMQ e JeroMQ

Autor:
Gustavo Braynner Carvalho

Orientador:
Nelson ROSA

26 de Junho de 2018

Universidade Federal de Pernambuco

Agradecimentos

A sorte de alguns é encontrar prazer em seu trabalho. Para mim, isto foi uma missão. Em todos os desafios que antecederam minha entrada no curso de Engenharia da Computação, pude estar feliz com o trabalho que realizava, mas não foi até metade dessa trajetória mais recente que consegui estar realizado com aquilo que escolhi como minha futura profissão. E por isso o meu primeiro agradecimento é à minha perseverança, minha sorte e a todos que me influenciaram a seguir em frente até aqui, pois hoje estou certo de que valeu a pena o esforço.

Dito isso, agradeço também a meu orientador Nelson Rosa pela chance de (e pelo apoio para) desenvolver esse trabalho conclusivo. Além disso, pelo interesse em meu progresso, pela paciência com meus passos em falso, e pelo conhecimento que me passou quando ainda era meu professor.

De todas as pessoas que tenho a agradecer, aquela a quem devo mais agradecimentos é meu pai, Jonas. Principal responsável pela minha educação, e o segundo personagem mais presente nessa trajetória. Sempre vibrando a meu lado, acreditando em mim, me ensinando e me apoiando.

À minha mãe, Klébia, agradeço pelo suporte emocional dado em todas as horas em que o pessimismo me abalava.

À minha parceira, Marina, agradeço pela compreensão em todas as horas que estive ausente, de corpo ou de cabeça, porque precisei focar em alcançar esse objetivo. Pelas inúmeras vezes que me ajudou com aquilo que era capaz, e pela sua dedicação em me ajudar a alcançar essa meta.

Agradeço também a meus vários mestres, que assim como meu orientador, me ensinaram e me inspiraram nesse trajetória em diversos momentos da minha vida acadêmica.

Agradeço por último a meus amigos. Embora muitas vezes tenham sido os responsáveis pelas distrações que mais complicaram minha vida, também foram eles que fizeram todo esse esforço valer a pena, pois é com eles que compartilho as melhores memórias de todos esses anos. As amizades que vieram desde antes do início dessa trajetória (Daniel, Manoel, Saulo e Vítor), aquelas que se iniciaram na metade do percurso (Breno, Bruno, Danilo, Gabriel, João Paulo, Marcelo, Pedro Costa e Pedro Leal), e aquelas adquiridas por causa do curso (André, Moiseis, Renê, Rodrigo e Tiago).

A todos, muito obrigado!

Resumo

Algumas das mais recentes evoluções na área de middleware são relacionadas ao desenvolvimento de middleware adaptativos. Uma forma de projetar sistemas de middleware adaptativos é com a utilização dos princípios de arquitetura de software. O middleware projetado dessa forma é visto como uma configuração de componentes e conectores. Uma forma de otimização desses middleware é o aumento no desempenho de seus componentes e conectores através da substituição de um componente ou conector por outro que cumpra a mesma função com funcionamento interno distinto. A proposta deste trabalho é desenvolver e avaliar conectores baseados em um serviço de filas (RabbitMQ) e conectores baseados em uma biblioteca de mensageria assíncrona (JeroMQ). Os experimentos feitos mostraram resultados favoráveis aos conectores baseados em RabbitMQ, com os conectores assíncronos possuindo maior vazão, e os conectores síncronos maior confiabilidade.

Palavras-chave: Middleware Adaptativo, Arquitetura de Software, Conectores, RabbitMQ, JeroMQ.

Abstract

Some of the most recent developments in the field of middleware are related to the development of adaptive middleware. In this context, a strategy to design adaptive middleware systems is to follow the principles of software architecture. A middleware designed in this way is seen as a configuration of components and connectors. An approach to optimize the middleware consists of replacing components/connectors by other elements having similar functionality but better performance. The purpose of this work is to develop and evaluate connectors based on a queue service (RabbitMQ) and an asynchronous messaging library (JeroMQ). The experimental evaluation shows that RabbitMQ performs better than JeroMQ, and the asynchronous connectors having a higher flow rate, and the synchronous connectors being more trustworthy.

Keywords: Adaptive middleware, Software Architecture, Connectors, RabbitMQ, JeroMQ.

Sumário

1 Introdução **7**
1.1 Contexto e Motivação . . . 7
1.2 Problema . . . 7
1.3 Objetivos . . . 7
1.4 Estrutura do Documento . . . 8

2 Conceitos Básicos **9**
2.1 Arquitetura de Software . . . 9
2.2 Conectores . . . 9
2.3 Middleware Adaptativo . . . 10
2.4 MidArch . . . 11
2.5 RabbitMQ . . . 11
2.6 JeroMQ . . . 11

3 Projeto e Implementaçaão de Conectores de Middleware **12**
3.1 Visão Geral . . . 12
3.2 Conectores baseados em RabbitMQ . . . 13
3.3 Conectores baseados em JeroMQ . . . 15
3.4 Conector assíncrono . . . 16
3.5 Conector síncrono . . . 18
3.6 Análise Comparativa . . . 21

4 Avaliação dos Conectores **23**
4.1 Objetivos . . . 23
4.2 Experimentos . . . 24
4.2.1 Tempo de espera do conector síncrono . . . 25
4.2.2 Confiabilidade . . . 28
4.2.3 Vazão . . . 30
4.2.4 Escalabilidade . . . 32

5 Conclusão e Trabalhos Futuros **35**
5.1 Conclusões . . . 35
5.2 Trabalhos futuros . . . 36

Referências Bibliográficas **37**

Lista de Figuras

2.1 Conector ligando nós A e B. 10
3.1 Rede de 3 nós com conectores baseados em RabbitMQ de fila exclusiva. 14
3.2 Conector baseado em RabbitMQ de filas compartilhadas entre nós A e C. . . . 14
3.3 Conector baseado em JeroMQ de filas compartilhadas entre nós A e C. 15
3.4 Digrama de fluxo bem sucedido no conector assíncrono. 17
3.5 Digrama de fluxo bem sucedido no conector síncrono. 19
3.6 Fluxo de *timeout* e desfazimento no conector síncrono. 21
4.1 Número de mensagens enviadas ao longo do tempo através de conector baseado em RabbitMQ assíncrono, conector baseado em RabbitMQ síncrono, conector baseado em JeroMQ assíncrono, conector baseado em JeroMQ síncrono. 32
4.2 Gráfico da vazão média do remetente em função do número de mensagens transmitidas através do conector baseado em RabbitMQ assíncrono para diferentes números de receptores. 33
4.3 Gráfico da vazão média do remetente em função do número de mensagens transmitidas através do conector baseado em RabbitMQ síncrono para diferentes números de receptores. 34

Lista de Tabelas

4.1 Vazão média (em mensagens por segundo) no envio de 100, 1 mil e 10 mil mensagens variando com o tempo de espera do conector baseado em RabbitMQ. 27
4.2 Taxa de reenvio no envio de 100, 1 mil e 10 mil mensagens variando com o tempo de espera do conector baseado em RabbitMQ. 27
4.3 Vazão média (em mensagens por segundo) no envio de 100, 1 mil e 10 mil mensagens variando com o tempo de espera do conector baseado em JeroMQ. . 28
4.4 Taxa de reenvio no envio de 100, 1 mil e 10 mil mensagens variando com o tempo de espera do conector baseado em JeroMQ. 28
4.5 Taxa de sucesso no envio de 100, 1 mil e 10 mil mensagens com cada modelo de conector. 29
4.6 Vazão média (em mensagens por segundo) de cada modelo de conector após 1 minuto. 31

1 Introdução

O foco deste trabalho é o desenvolvimento e avaliação de conectores baseados em RabbitMQ [1] e JeroMQ [2]. Este capítulo descreve inicialmente o contexto e a motivação do trabalho. Em seguida, é descrito o problema tratado e os objetivos traçados no início do estudo. A última parte desta introdução descreve a estrutura desta monografia.

1.1 Contexto e Motivação

Middleware adaptativos tem sido projetados com a utilização de princípios de arquitetura de software. Esses princípios englobam a modularidade das partes (módulos de propósito singular) [3], funcionamento unitário e independente de condições externas, e minimização da repetição de código. Esse tipo de middleware pode ser visto como uma montagem de diferentes componentes e conectores interconectados.

No middleware adaptativo, componentes e conectores podem ser substituídos por módulos similares [4]. Essa substituição é por um gerente de adaptação do próprio middleware. Para pode substituir módulos após mudanças de cenário, o middleware precisa ter à sua disposição uma diversidade componentes e conectores.

Esses componentes e conectores devem ser desenvolvidos isoladamente, e testados de maneira unitária. Se um componente ou conector for corretamente implementado em relação às regras de acoplamento fraco iniciais, o mesmo deve funcionar corretamente aplicado em diferentes sistemas que o utilizam. Middleware costumam ter vários conectores. A reutilização do mesmo módulo previne repetição de código.

1.2 Problema

Como sistemas de middleware adaptativos são uma configuração de componentes e conectores, o seu desempenho é diretamente afetado pelo desempenho destes elementos. O desenvolvimento de novos componentes e conectores é uma possibilidade de melhora no desempenho do middleware. Este estudo foca no problema de projetar conectores de middleware.

Conectores para middleware adaptativos precisam ser desenvolvidos de acordo com os princípios de arquitetura de software. No desenvolvimento desses conectores é necessária a implementação de uma forma de comunicação entre os extremos de um conector. Essa forma de comunicação pode utilizar uma variedade de tecnologias para o seu desenvolvimento, como funções básicas de comunicação remota da linguagem, serviços de filas, bibliotecas de mensageria e assim por diante.

Para saber quais tecnologias oferecem uma melhor solução para esse problema, é necessário avaliar o impacto de cada uma dessas opções. Este estudo foca no funcionamento de conectores baseados em algumas dessas soluções de comunicação em diferentes cenários de utilização.

1.3 Objetivos

O foco deste estudo é o desenvolvimento e avaliação de conectores que podem ser usados em middleware adaptativos. Estes conectores devem seguir os princípios de arquitetura de software em suas implementações.

Para o desenvolvimento dos conectores, foram utilizados o serviço de filas RabbitMQ [1] e a biblioteca de mensageria assíncrona *open-source* JeroMQ, que é uma implementação Java da biblioteca ZeroMQ [5]. A diferença entre esses conectores é apenas no método de comunicação utilizado pelo conector para transmitir mensagens entre suas pontas.

Para cada tecnologia de comunicação, foi produzido um modelo de conector síncrono e um modelo de conector assíncrono. As implementações dos modelos de sincronia são análogas, e variam apenas no uso das funções de envio e recebimento de mensagens.

A meta deste estudo é produzir e avaliar esses conectores. A partir de comparações internas, apontar qual tecnologia esteve associada aos melhores resultados, e definir parâmetros que devem ser utilizados no momento da escolha da solução mais apropriada ao problema em questão.

1.4 Estrutura do Documento

No Capítulo 2 são apresentados os conceitos básicos que precedem este estudo e uma descrição do estado atual dos estudos nas áreas de conectores, middleware adaptativos e um resumo sobre o MidArch. É importante observar que o MidArch, um middleware baseado em conceitos de arquitetura de software, serviu como base para este estudo.

O Capítulo 3 descreve e analisa as decisões de projeto feitas para a construção de cada conector. As seções deste capítulo analisam os conectores a partir da fixação de apenas uma variável (sincronia ou tecnologia de comunicação), dado que as mudanças provocadas por essas características são ortogonais entre si.

No Capítulo 4, os conectores desenvolvidos são avaliados. Antes dos testes, as definições necessárias para criar uma possibilidade de comparação são feitas, assim como também são definidos parâmetros e desenho de cada teste. Em sua maioria, cada tipo de teste foi executado da mesma forma para todos os conectores, e os resultados obtidos mostram uma comparação entre os conectores. Os conectores são comparados em relação a cada característica variante entre eles.

Concluindo este trabalho, apontamos qual conector performou melhor em cada contexto e em relação a cada aspecto, além de identificar os cenários onde a utilização de cada conector seria a mais apropriada. Também foi analisada a possibilidade de trabalhos futuros que podem ser derivados deste estudo. Isto está contido no capítulo 5.

2 Conceitos Básicos

O objeto central deste estudo é a avaliação de algumas alternativas de conectores utilizando o serviço RabbitMQ e JeroMQ com o propósito de que tais conectores possam ser usados em middlewares adaptativos. Esses tipo de middleware é implementado seguindo princípios de arquitetura de software. Neste capítulo estão definições e análises dos avanços nas áreas de arquitetura de software, middleware adaptativo e conectores. No fim deste capítulo, há um resumo sobre o MidArch, o RabbitMQ e o JeroMQ.

2.1 Arquitetura de Software

Os princípios de arquitetura de software no desenvolvimento de sistemas já haviam sido mencionados antes, mas a sua melhor definição é encontrada no estudo feito por Perry e Wolf, que definiu um modelo de arquitetura de software como constituído por elementos, forma e racional [6]. Arquitetura ficou então definida como uma base para a satisfação de requisitos, estimativas de complexidade, reuso e análise de consistência de softwares.

O conceito de arquitetura de software modificou a maneira de se chegar ao projeto de uma solução. A estratégia de separar para conquistar no desenvolvimento de soluções passou a ser o novo padrão. A divisão do trabalho deixou de acontecer na fase de projeto da solução e passou a ser feita na parte de divisão do problema. Com problemas unitários, várias soluções mais simples podiam ser implementadas e utilizadas conjuntamente para produzir a solução do problema original. Isto não apenas diminuiu a complexidade das soluções, como também aumentou a testabilidade da mesma. Outros benefícios da utilização dos princípios de arquitetura de software são: a possibilidade de reuso de código, através do reuso de componentes; a facilitação da manutenção de um sistema desenvolvido com esses princípios, onde as correções podem ser exclusivas a um módulo, e não envolverem a solução por completo. O desenvolvimento de um sistema que segue os princípios de arquitetura de software deve basear seu projeto através dos requisitos do sistema [7].

Num sistema desenvolvido de acordo com os princípios de arquitetura de software, componentes e conectores são elementos de primeira classe [8]. Por esse motivo, um sistema desenvolvido com os princípios de arquitetura de software pode ser visto como uma configuração específica de componentes e conectores.

2.2 Conectores

Com a crescente complexidade dos softwares e o surgimento de técnicas de programação com componentes, surguiu o conceito de conectores: um pedaço de software modular responsável pela comunicação entre duas pontas [9]. Conectores desempenham 3 funções fundamentais: inibição de ação, sincronização de ação e transmissão de mensagem [9]. Conectores provêem as ligações entre os componentes da arquitetura do software. Eles são entidades explicitamente arquiteturais que unem componentes e mediam ações e a comunicação entre os mesmos [10].

Avanços no grau de modularidade é um objetivo muito buscado, e há algum tempo já possuímos vários exemplos de conectores que provêem mensageria como um serviço [11]. Mas, com o surgimento da demanda por conectores mais complexos, o custo da personalização se tornou proibitivo. Para resolver esse problema, foram desenvolvidas técnicas de montagem de conectores baseadas em reutilização de soluções genéricas [12].

Conectores são utilizados em middleware para ligar componentes [13]. A comunicação provida pelos conectores é a base para que a construção do middleware possa ser feita a partir da montagem de componentes e conectores. O conector também é similar aos componentes no sentido de que é uma peça modular e substituível do middleware. O conector pode ser compreendido como um componente de primeira classe [9].

Em [13], temos a chance de ver exemplos de conectores que ligam um número indefinido de componentes, estabelecendo relações mais complexas entre os mesmos. O desenho da forma mais básica de conector pode ser visto na figura 2.1, onde são representados os componentes A e B e suas duas conexões a um conector: seu canal de envio de mensagens e seu canal de recebimento de mensagens.

Figura 2.1: Conector ligando nós A e B.

2.3 Middleware Adaptativo

Middlewares são softwares utilizados em sistemas distribuídos. Eles são compostos de componentes e conectores modulares orientados a serviço que seguem padrões de projeto bem estabelecidos [14]. Middleware adaptativo é um tipo específico de middleware capaz de modificar o seu funcionamento em tempo real [4]. Essas modificações advêm de várias rotinas de inspeção, e são a base de muitos sistemas complexos atuais que requerem interoperabilidade e ciência do próprio contexto [15]. O objetivo do middleware pode ser definido como permitir a interação entre diferentes softwares [14]. Um dos desafios comuns na busca desse objetivo é possibilitar a interoperabilidade em redes heterogêneas [3]. A demanda por soluções genéricas aumenta o nível de complexidade desses softwares. Soluções que gerenciam complexidade tentam esclarecer e facilitar a configuração de middlewares [16].

Um dos questionamentos fundamentais em relação à produção de middleware adaptativo é como o mesmo tomará a decisão de se adaptar [4]. A resposta para esse problema não é uma só, e uma variedade de soluções já foi apresentada em relação a como construir algoritmos de decisão, como mineração de processos [17].

Requisitos de um middleware adaptativo não se resumem à auto-adaptação. Também é necessário que o software atenda aos requisitos de dependabilidade, heterogeneidade e robustez [18]. Essas características permitem que visualizemos o middleware como um serviço [15].

Atualmente, a utilização de middlewares adaptativos tem se tornado ubíqua. Os avanços mais recentes incluem middlewares adaptativos em dispositivos móveis [19] e sua utilização em outros dispositivos conectados à internet (Internet das Coisas) [20].

A maioria dos desenvolvimentos mais recentes na área de middleware adaptativo focam no uso de tecnologias existentes para adicionar adaptabilidade a novos domínios de aplicações [17].

2.4 MidArch

Middleware adaptativos surgiram para resolver o problema da aplicação de um middleware em um cenário que não é completamente compreendido pelos desenvolvedores ou para um cenário que pode sofrer mudanças não antecipadas no desenvolvimento do middleware. Através de adaptação, o middleware poderia se adequar a estes cenários [21]. Essa adaptação viria através de verificação contínua do desempenho do middleware, que tomaria decisões capazes de fazê-lo evoluir.

O MidArch é uma solução de ponta-a-ponta com o propósito de auxiliar desenvolvedores na implementação e execução de middleware adaptativo a partir de arquitetura de software [22]. O MidArch fornece, durante o desenvolvimento, o uso de métodos formais para o desenvolvimento desses sistemas, sem envolver a complexidade de tais métodos no projeto do usuário. Durante a execução do middleware desenvolvido, o MidArch utiliza suas ferramentas no gerenciamento da execução do middleware e suas adaptações.

A motivação para a utilização do MidArch é variabilidade dos mecanismos de adaptação. A meta do MidArch é evitar comportamento indesejados, que podem surgir a partir de mudança nos requisitos da aplicação ou nas condições do ambiente. O processo de avaliação e adaptação do middleware é contínuo, e pode acontecer a qualquer momento. As mudanças consistem da substituição de componentes e conectores por módulos diferentes que cumpram a mesma função [22].

2.5 RabbitMQ

O RabbitMQ é um intermediário para mensageria. O serviço é constituído de uma plataforma comum que permite o envio e recebimento de mensagens por parte das aplicações [23]. A tecnologia oferece o compartilhamento de filas como serviço. A característica de uma rede interligada pelo RabbitMQ é centralizada.

O serviço oferece garantias de comunicação, como confirmação de recebimento e de entrega. Os filtros do conteúdo das mensagens em cada fila são customizáveis. A customização pode ser feita por tipo de mensagem, tópico, e assim por diante [23].

2.6 JeroMQ

O JeroMQ é uma implementação puramente Java da biblioteca ZeroMQ [2]. O ZeroMQ é uma biblioteca de mensageria de alto desempenho. Os padrões de conexão oferecidos por essas bibliotecas incluem *pub-sub*, *push-pull* e *router-dealer*.

O projeto JeroMQ é open-source, e mantém a característica de alto desempenho da biblioteca ZeroMQ contendo todas as suas funcionalidades [2].

3 Projeto e Implementaçaão de Conectores de Middleware

O elemento de middleware adaptativo que é o foco deste estudo é o conector. O nosso objetivo é o desenvolvimento e a avaliação de novos conectores assíncronos e síncronos baseados em tecnologias de mensageria existentes.

Nesta seção encontramos a descrição dos conectores desenvolvidos para o propósito deste estudo. A escolha pela tecnologia dos conectores que foram serão implementados e as características desses conectores (os requisitos que devem atender), são especificados na Seção 2.1. Nas outras seções estão os detalhamentos do que cada escolha ou característica deve atribuir a um conector, os projetos e a descrição da implementações.

3.1 Visão Geral

Para a produção de middlewares adaptativos, é necessário o uso de componentes modulares [3]. Componentes modulares possuem características genéricas de entrada e saída (acoplamento fraco), o que os torna substituíveis uns pelos outros [24]. Nosso objetivo é a comparação de conectores candidatos a fazerem parte de um middleware adaptativo, e para isso os mesmos devem possuir as mesmas entradas e saídas.

Um conector é responsável pela transmissão de mensagens ponta-a-ponta [13]. A forma mais básica de uma mensagem digital é a binária. No momento do envio de uma mensagem, a informação necessária ao remetente é a *quem* enviar e *o quê* enviar. Do ponto de vista do destinatário, é importante saber de quem é a mensagem, e qual o seu conteúdo (o quê).

Os cenários sob os quais analisamos cada tipo de conector envolviam apenas um remetente. Dessa maneira, é possível especificar filas por destinatário. Redes mais complexas, porém, possuem nós que atuam tanto como remetente, quanto como destinatário. Dessa forma, uma escolha deve ser feita durante o projeto do conector: se o mesmo fará uso de duas filas exclusivas (de A para B e de B para A) sem identificação de remetente necessária, ou se cada nó receberá todas as suas mensagens pela mesma fila, e a mensagem conterá identificação do remetente.

Do ponto de vista do receptor (destinatário), a mensagem deve ser obtida através de uma solicitação de leitura. Essa leitura faz com que, de maneira atômica, a mensagem seja lida pelo componente diretamente conectado à saída do conector, e desenfileirada na fila utilizada. No caso do uso do conector de filas compartilhadas que se deseja receber uma mensagem de um remetente específico, é necessário o uso de filas locais que sejam diferenciadas pelos remetentes das mensagens que a mesma contém.

A implementação de um conector pode envolver a utilização de um serviço de filas, que age como intermediário e centraliza as conexões entre os nós de uma rede. A outra opção é a utilização de filas sem auxílio de intermediários, criadas pelos próprios nós em cada ponta do conector. A utilização de um serviço de filas simplifica a arquitetura de um projeto, além de possuir controles previamente implementados pelo criador do serviço de filas. Ao implementar conectores descentralizados, fica a cargo do desenvolvedor implementar mecanismos de controle de fluxo e confiabilidade. A vantagem que se espera obter do conector descentralizado é no quesito desempenho, dado que o mesmo requer um menor número de passos para a conclusão de seu fluxo.

Para a implementação de um conector que faz uso de um serviço de filas, o serviço de filas escolhido foi o RabbitMQ [1]. Entre os serviços de fila open-source, o RabbitMQ é o mais amplamente usado e documentado na literatura da área. RabbitMQ é um serviço de filas *open-*

source implementado em Erlang com mais de 10 anos desde sua primeira versão. O RabbitMQ é um *broker* de mensagens open-source que oferece serviço de confirmação de entrega de mensagens e enfileiramento flexível que utiliza o protocolo AMQP (*Advanced Message Queuing Protocol*) [25] [23].

Para a implementação de um conector descentralizado, utilizamos a biblioteca de mensageria assíncrrona [2]. JeroMQ é a implementação em Java da biblioteca de mensageria assíncrona de alto desempenho *open-source* chamada ZeroMQ [25]. Essas bibliotecas de mensageria assíncrona oferecem comunicação de alta velocidade. O ZeroMQ é a biblioteca de mensageria assíncrona mais bem-documentada, e tem estado sob constante atualização há mais tempo que outras bibliotecas de mensageria assíncrona *open-source*. A escolha pela implementação Java se deu pela facilidade da integração.

Ambas as tecnologias utilizadas para a implementação dos conectores possuem seus próprios mecanismos de aumento de confiabilidade, com limitações características do caráter distribuído ou não de cada uma delas. A implementação mais simples de ambos os conectores se trata da sua versão assíncrona, dado que um mecanismo de sincronia precisaria ser acrescentado sobre o conector assíncrono para a obtenção de sua versão síncrona. Conectores síncronos possuem vantagens em relação aos conectores assíncronos nos quesitos confiabilidade e sincronicidade. Por esse motivo, também analisamos versões síncronas dos conectores baseados em RabbitMQ e JeroMQ.

3.2 Conectores baseados em RabbitMQ

O RabbitMQ é um serviço de filas. Um conector requer 2 filas, nas quais um dos nós envia mensagens por uma e recebe mensagens pela outra, e sua contra-parte faz o oposto. Essas filas podem ser exclusivas por relação remetente-destinatário, ou a mesma pode ser específica por destinatário, e o remetente se identifica em um cabeçalho de sua mensagem. A escolha entre uma dessas duas implementações deve ser feita baseado no número de filas necessárias e no tamanho máximo de cada fila.

Partindo de uma implementação que contém filas exclusivas para cada relação remetente-destinatário, uma rede de N nós requer N*(N-1) filas, o que é o dobro do número de conectores. Nessa mesma implementação, uma fila só poderia ser sobrecarregada caso um único remetente enviasse repetidas mensagens a um único receptor incapaz de consumí-las.

A figura 3.1 ilustra uma rede de 3 componentes interligadas por conectores baseados em RabbitMQ de fila exclusiva. Cada conector é constituído por duas filas, com um componente conectado ao conector podendo enfileirar mensagens em uma fila, e desenfileirá-las na outra, e o inverso sendo verdadeiro para o outro componente. As filas estão localizadas no serviço do RabbitMQ. A rede com esses conectores requer a existência de 6 filas.

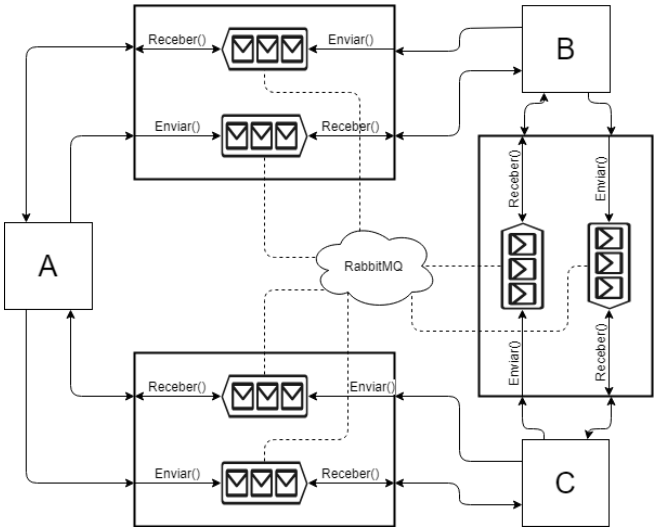

Figura 3.1: Rede de 3 nós com conectores baseados em RabbitMQ de fila exclusiva.

Partindo de uma implementação que contém filas exclusivas para cada destinatário, uma rede de N nós requer N filas. Neste sentido, a rede é muito mais escalável em relação ao número de nós. Em relação ao tamanho da fila, é possível que a mesma seja sobrecarregada num cenário em que vários remetentes em conjunto enviassem um total de mensagens superior ao limite da fila antes que o receptor fosse capaz de consumí-las. A possibilidade de uma fila ser sobrecarregada é maior no conector de filas compartilhadas. A figura 3.2 ilustra o exemplo de um conector baseado em filas compartilhadas, com as componentes B e C sendo capazes de enviar mensagens para a mesma fila, utilizada por A para receber mensagens.

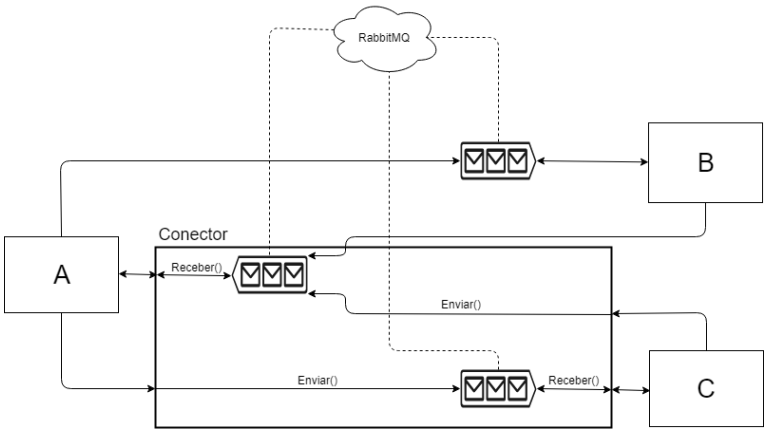

Figura 3.2: Conector baseado em RabbitMQ de filas compartilhadas entre nós A e C.

O *trade-off* entre ambos os cenários deve levar em consideração o que é mais crítico entre

o tamanho máximo de uma fila e o número máximo de filas. No caso do RabbitMQ, o serviço é robusto o suficiente para suportar qualquer uma das implementações. Por questões de complexidade de implementação e paralelismo com a implementação do JeroMQ, optamos pelo modelo de fila única por receptor. A opção pelo modelo de fila compartilhada entre remetentes faz com que seja necessário a adição da identificação do remetente no cabeçalho da mensagem para que o receptor seja capaz de identificá-lo.

Conectores baseados em RabbitMQ podem ser implementados em uma variedade de linguagens [26]. Os conectores desenvolvidos são bilateralmente simétricos em funcionalidades, e um conector pode ser constituído por 2 metades implementadas em linguagens diferentes que compartilhem as mesmas filas. Esse tipo de acoplamento permite que o desenvolvedor de uma ponta do conector não precise se preocupar com a maneira a qual foi implementada a outra ponta, desde que a mesma trate os dados que são recebidos e que são enviados da mesma forma.

3.3 Conectores baseados em JeroMQ

JeroMQ é a implementação Java da biblioteca de mensageria assíncrona ZeroMQ. Diferentemente do RabbitMQ, as filas de um conector baseado em JeroMQ ficam localizadas no receptor das mensagens. A biblioteca adiciona uma camada de abstração à comunicação entre *sockets* dos nós. O remetente envia sua mensagem para uma porta do receptor, que por sua vez deve estar ouvindo naquela mesma porta e desenfileirando as mensagens.

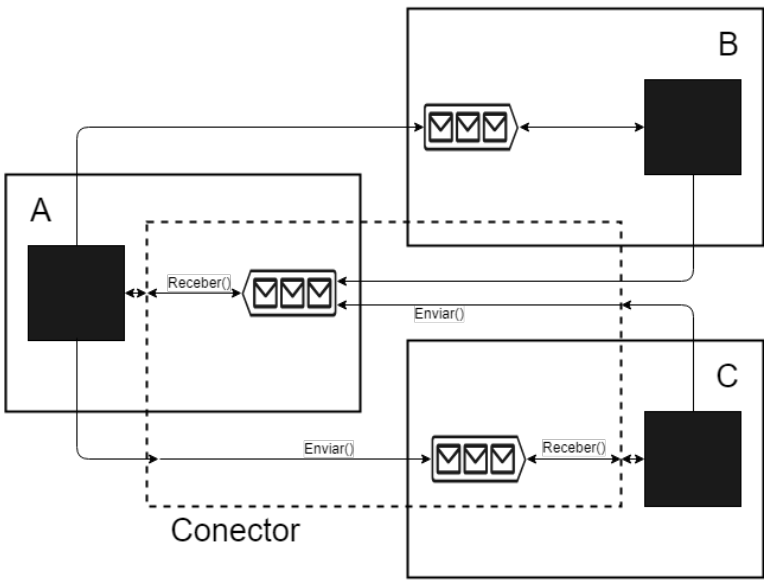

Figura 3.3: Conector baseado em JeroMQ de filas compartilhadas entre nós A e C.

A figura 3.3 ilustra o exemplo de um conector JeroMQ baseado em filas compartilhadas. As componentes B e C enviam mensagens para a mesma fila, localizada no ambiente da componente A, que desenfileira mensagens dessa fila.

Assim como nos conectores baseados em RabbitMQ, é possível definir filas exclusivas por relações remetente-destinatário. Mas o modelo mais intuitivo é constituído de filas exclusivas por destinatário, uma vez que o mesmo deve verificar a presença de mensagens nas portas utilizadas pelo conector. A decisão por esse modelo também simplifica a implementação, uma vez que a implementação do conector pode assumir que ambas as partes estarão escutando na mesma porta sempre, e o remetente pode enviar mensagens sem negociar quais portas serão utilizadas para a comunicação em cada uma das partes.

Assim como no conector que faz uso do serviço de filas, dada a escolha pelo modelo de fila exclusiva por receptor, é necessária a presença da identificação do remetente no cabeçalho da mensagem que permite ao receptor identificar o remetente.

JeroMQ é uma biblioteca Java e a linguagem utilizada para implementar o conector também deve ser. Mas vale ressaltar que JeroMQ é uma versão pura em Java da biblioteca ZeroMQ, a qual pode ser importada em qualquer linguagem, o que possibilitaria a criação de conectores através do acoplamento de implementações das mesmas funcionalidades em linguagens diferentes.

3.4 Conector assíncrono

A implementação mais simples de um conector é a sua versão assíncrona. Esse conector é constituído de duas filas: uma para qual um nó envia a mensagem e sua contra-parte a recebe e outra em que o oposto acontece. No conector assíncrono não existe ligação temporal entre o envio e o recebimento de uma mensagem além de que o recebimento da mensagem pode acontecer em qualquer momento após a conclusão do seu envio. A comunicação é bem sucedida quando o receptor consegue desenfileirar a mensagem enviada pelo remetente.

Externamente ao conector, os componentes devem ser capazes de utilizar duas funcionalidades que correspondem ao envio e recebimento das mensagens. O conector deve ser inicializado em duas partes, sendo elas as metades simétricas adjacentes a cada um dos componentes conectados pelo conector. Nessa inicialização, cada metade deve estabelecer um canal para o envio de suas mensagens e abrir seu canal para o recebimento de mensagens.

Quando o componente conectado ao conector solicita o envio de uma mensagem, a mesma deve ser encapsulada em um formato particular ao conector. Esse formato deve conter um cabeçalho e a mensagem em sua forma original ou codificada. O cabeçalho dessa mensagem deve conter informações como a identificação do remetente e a identificação da mensagem. A necessidade da presença da identificação do remetente surge da utilização de filas exclusivas ao receptor, mas compartilhada por remetentes. A utilização de identificação da mensagem é opcional no conector assíncrono, uma vez que o mesmo não possui mecanismo interno de reenvio de mensagens, e uma mensagem cujo componente solicita o seu envio uma única vez pode ser recebida pelo receptor apenas uma vez (ou nenhuma, no cenário de falha de comunicação). A codificação da mensagem também é opcional. Essa codificação pode ser estática (podendo ser decodificada por qualquer recipiente da mesma) ou dinâmica (cenário no qual parâmetros da codificação devem ser negociados em uma etapa de handshake da inicialização do conector).

Uma escolha de projeto a ser feita na implementação das partes receptoras do conector é em relação à localização da fila do receptor. No caso do conector baseado em serviço de filas, é se as mensagens enviadas devem se acumular no serviço de filas ou localmente. No caso do conector que faz uso de uma biblioteca de mensageria assíncrona, se as mensagens

enviadas devem se acumular na porta utilizada pelo receptor ou localmente. A implementação mais simples desenfileira as mensagens apenas no momento da solicitação de recebimento pelo componente externo. A outra opção conta com uma rotina de desenfileiramento das mensagens e reenfileiramento em uma fila local, da qual a mensagem deve ser novamente desenfileirada durante a solicitação de recebimento.

A Figura 3.4 apresenta um cenário em que o receptor solicita ao conector o recebimento de uma mensagem anterior ao envio da mesma pelo outro componente, o que ocasiona em falha no seu recebimento. Em seguida, o outro componente solicita o envio da mensagem ao conector. Finalmente, o receptor solicita o recebimento da mensagem, que dessa vez é bem-sucedido.

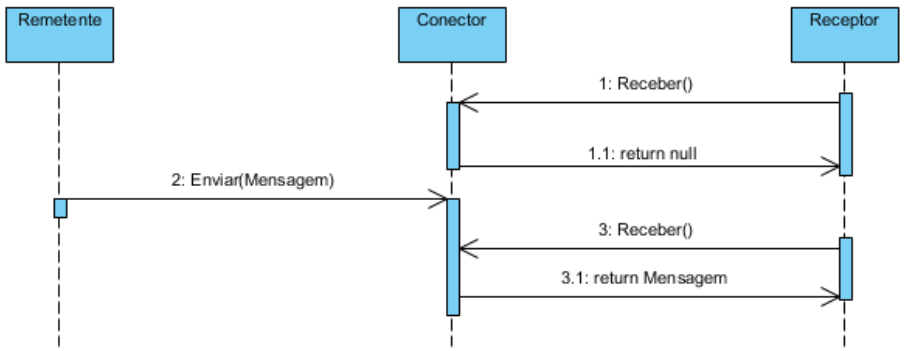

Figura 3.4: Digrama de fluxo bem sucedido no conector assíncrono.

Todos os eventos de falha de comunicação que podem ocorrer são internos ao conector. No caso de um conector baseado em serviço de filas, existem dois possíveis cenários de falha de comunicação. O primeiro é no momento do envio, no qual a mensagem não chega ao serviço de filas e consequentemente não chega ao receptor. O segundo cenário é aquele em que a falha ocorre no desenfileiramento da mensagem no serviço pela parte receptora do conector. Nesse cenário, a mensagem seria recebida com sucesso na próxima tentativa de recebimento bem-sucedida, o que torna esse cenário menos grave. No caso de um conector baseado em uma biblioteca de mensageria assíncrona, o único cenário de falha de comunicação ocorre em uma falha no seu envio.

Em relação aos conectores deste estudo, tanto aqueles baseados em serviço de filas quanto aqueles baseados em biblioteca de mensageria assíncrona utilizam rotinas de desenfileiramento das mensagens recebidas para uma fila local. Essa opção foi feita com o intuito de manter as mensagens em um ambiente mais estável e sob maior influência do desenvolvedor. A rotina de desenfileiramento da porta do receptor pode iterar irrestritamente, uma vez que uma solicitação de desenfileiramento de uma mensagem na porta prévia ao envio da mesma interrompe a rotina até que uma mensagem seja recebida. No caso dessa rotina referente ao conector baseado em serviço de filas, existe uma necessidade ainda menor de tratamento, dado que a biblioteca compatível com a utilização do serviço inicia o consumo da mensagem assim que a mesma se faz presente na fila do receptor, solicitando apenas que o desenvolvedor implemente apenas a rotina a ser executada no momento do seu recebimento.

O formato das mensagens destes conectores assíncronos contém um cabeçalho formado

pelos identificadores do remetente e da própria mensagem, e a mensagem em sua forma original, sem codificação.

O pseudo-código abaixo ilustra a rotina de execução da parte receptora do conector assíncrono, encarregada pela transferência da mensagem da fila compartilhada para uma fila local.

```
byte[] message = NextMessage();

String str_message = ByteArrayToString(message);

String senderId = SenderFromMessage(str_message);
String content = ContentFromMessage(str_message);
Enqueue(senderId, messageId, content);
```

Essa rotina é iterada repetidamente até a *destruição* do conector. O método que solicita a próxima mensagem gera um estado de interrupção até a chegada de uma nova mensagem. Essa rotina é iterada em paralelo ao funcionamento do resto do conector.

O funcionamento dos conectores, então, pode ser resumido por: o encapsulamento da mensagem original, seu envio através do seu mecanismo de comunicação específico, o recebimento através deste mecanismo, o desencapsulamento no lado receptor, o enfileiramento local, e seu final desenfileiramento no momento da solicitação de recebimento. Ambas as implementações foram feitas em Java SE 8.

3.5 Conector síncrono

Em cenários onde há a necessidade de sincronicidade e/ou consistência entre componentes, é necessária a utilização de um conector síncrono. O conector síncrono possui os mesmos requisitos funcionais do conector assíncrono, com a adição de um requisito de sincronicidade. A exigência no conector síncrono é que o remetente conclua a etapa de envio apenas quando o receptor concluir o recebimento da mensagem. A comunicação é concluída como bem sucedida quando o remetente é informado do sucesso do receptor no recebimento da mensagem.

O mecanismo que permite a existência de sincronicidade no conector é a utilização de mensagens de reconhecimento (ack) [27]. Ao receber uma mensagem, a rotina executada no conector deve enviar à parte receptora oposta um ack, informando o recebimento bem-sucedido da mensagem.

A inicialização no conector síncrono é similar a do conector assíncrono, com a diferença presente no tratamento do recebimento de mensagens. O conector também deve conter mecanismos que lidam com cenários de falha de comunicação.

O cabeçalho da mensagem enviada (encapsulamento da mensagem original), requer a presença de identificação de mensagens de forma a evitar que o receptor aceite mensagens repetidas. Essas duplicatas podem surgir num cenário de falha de comunicação no envio do *ack*. Além disso, o cabeçalho também deve conter a identificação do remetente, uma vez que o conector faz uso de fila exclusiva por receptor e compartilhada por remetentes. Assim como no conector assíncrono, a codificação da mensagem original é opcional.

O tempo necessário para o receptor desenfileirar a mensagem após seu recebimento e enviar o *ack* para o remetente se traduz em tempo ocioso e blocante para o remetente, o que faz com que seja importante que o receptor esteja sempre checando sua fila de mensagens. Como a

forma de saber se há uma mensagem em sua fila e quem é o remetente da mesma é desenfileirando a mensagem da fila compartilhada. Após o desenfileiramento, a mensagem precisa ser reenfileirada localmente. Por esse motivo, o conector síncrono requer a existência de filas locais. A existência de uma rotina de transferência de mensagens da fila de comunicação para a fila local é obrigatória. Assim como em uma das implementações de conector síncrono, as solicitações de recebimento fazem com que o conector faça um *pop* na fila local.

A Figura 3.5 apresenta o fluxo de um cenário em que o receptor solicita ao conector o recebimento de uma mensagem anteriormente ao envio da mesma pelo outro componente, o que ocasiona uma falha no seu recebimento. Em seguida, o outro componente solicita o envio da mensagem ao conector e espera pela mensagem de reconhecimento. O conector responde com a mensagem de reconhecimento e conclui a participação do remetente. Finalmente, o receptor solicita o recebimento da mensagem que é bem-sucedido.

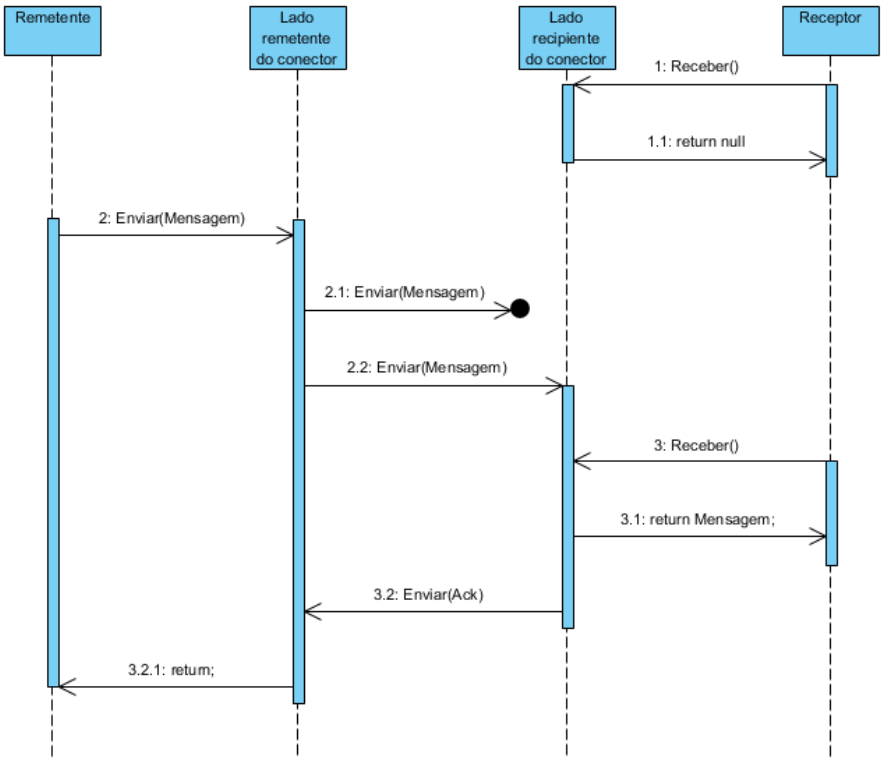

Figura 3.5: Digrama de fluxo bem sucedido no conector síncrono.

A implementação do conector síncrono é mais complexa que a do conector assíncrono devido à necessidade de mecanismos de sincronicidade. O primeiro desses mecanismos é o envio de mensagens de *ack* na rotina de recebimento de uma mensagem. Mas como não é transparente ao remetente se a falha ocorreu durante o envio da mensagem original ou durante o envio do *ack* pela parte receptora, após um certo intervalo sem receber o *ack* do receptor, o remetente

deve enviar a mensagem novamente. Isso deve se repetir até que o mesmo receba o *ack*. O último problema com o qual é necessário lidar é o da presença de duplicatas nos casos de falha do envio do *ack*. Para lidar com esse problema, o receptor deve manter um histórico de identificadores de mensagens no qual ele pode conferir a presença ou não do identificador da mensagem atual. Se o identificar estiver presente, a mensagem é uma duplicata. Caso contrário, é uma nova mensagem e o identificador dessa mensagem deve ser adicionado ao histórico. O fluxo desse conector síncrono deixa a parte remetente encarregada de garantir a transmissão da mensagem. Esse fluxo contém 2 passos de comunicação.

Dado que a utilização de filas locais é obrigatória, assim como a presença de identificadores de mensagem e remetente no cabeçalho das mensagens, o conector síncrono é similar em todos esses aspectos ao conector assíncrono. Outra similaridade é a repetição da escolha pelo não-uso de codificação para a transmissão da mensagem original.

O funcionamento do conector síncrono segue as seguintes etapas:

1. Empacotamento da mensagem original;
2. Envio da mensagem completa através do mecanismo de comunicação específico;
3. Desenfileiramento da mensagem completa;
4. Desempacotamento da mensagem completa;
5. Envio da mensagem de ack pelo receptor;
6. Recebimento da mensagem de ack pelo remetente; e
7. Desenfileiramento local da mensagem original no momento da solicitação de recebimento.

Assim como o conector síncrono, esse conector síncrono também foi implementado em Java SE 8.

O pseudo-código abaixo ilustra a rotina de execução da parte receptora do conector síncrono, encarregada pela transferência da fila compartilhada para uma fila local e pela solicitação de envio da mensagem de recebimento após o desenfileiramento.

```
byte[] message = NextMessage();

String str_message = ByteArrayToString(message);

String messageId = IdFromMessage(str_message);
if (IsAck(str_message)) {
  StopSending(messageId);
} else {
  String senderId = SenderFromMessage(str_message);
  String content = ContentFromMessage(str_message);
  Enqueue(senderId, messageId, content);
  WaitForDequeuingAndSendAckAsync(senderId, messageId);
}
```

A maneira como essa rotina é iterada e o funcionamento do método que solicita a próxima mensagem são idênticos nos conectores síncrono e assíncrono.

Para evitar cenários de *deadlock* com um componente receptor ocioso, o conector síncrono deve limitar o número de tentativas de reenvio da mensagem. É possível que uma inconsistência surja, caso o receptor receba a mensagem e o remetente não receba essa informação antes de desistir do envio da mensagem. Se esse cenário não for aceitável, é necessária a existência de um tipo de mensagem de desfazimento, que deve ser enviada pela parte remetente no caso do recebimento de um *ack* referente à transmissão mal-sucedida.

Um fluxo com mensagem de desfazimento é ilustrado na figura 3.6. Neste fluxo, o conector responde a mensagens de ack de transmissões mal-sucedidas com a solicitação do desfazimento das ações realizadas por causa da mensagem desta transmissão.

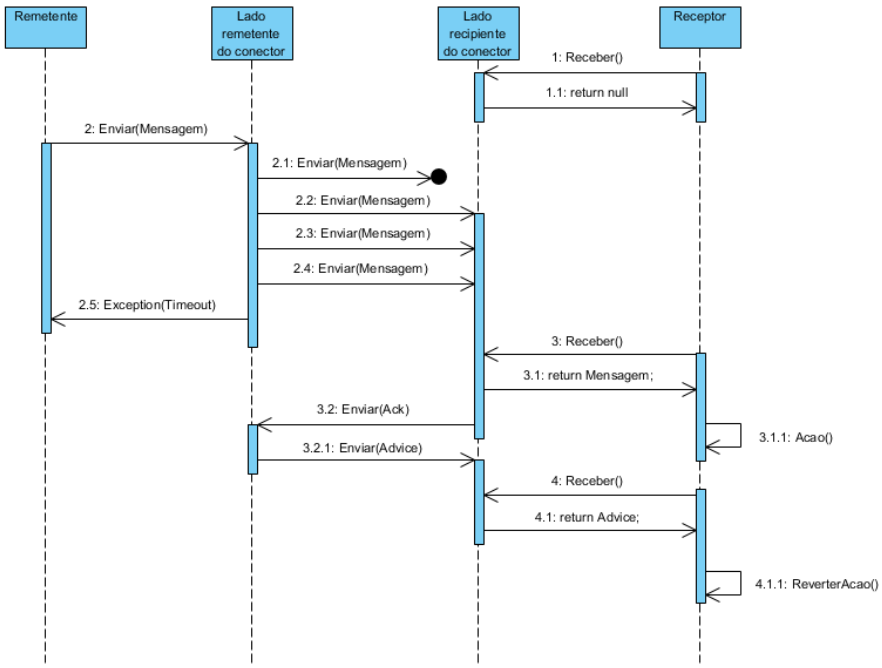

Figura 3.6: Fluxo de *timeout* e desfazimento no conector síncrono.

3.6 Análise Comparativa

Neste capítulo foram descritos modelos de conectores que podem divergir em dois aspectos diferentes: a tecnologia que é utilizada na comunicação entre as partes (serviço de filas ou biblioteca de mensageria assíncrona) e a presença ou ausência de mecanismos de sincronicidade. O total de conectores possíveis derivados da permutação entre essas opções é quatro. As diferenças de implementação entre conectores causadas pela tecnologia e as diferenças causadas pela sincronicidade, no geral, são ortogonais. É importante observar na análise quantitativa o quanto essa característica é verdade em relação ao desempenho.

As diferenças decorrentes da escolha da tecnologia a ser utilizada são, até certo ponto, incógnitas. O motivo para isso é que a característica de cada conector depende muito da

implementação das próprias tecnologias. Sendo assim, por ser uma forma de comunicação direta entre pares, a tendência é de um melhor desempenho por parte do conector baseado em biblioteca de mensageria assíncrona. Em relação a suas implementações, os conectores pouco diferem entre si.

Para o componente receptor há pouca diferença entre os conectores síncrono e assíncrono. O mesmo pode receber a sua mensagem logo após a mesma ser recebida por sua parte do conector. É importante, porém, que o componente receptor seja parte ativa da comunicação. No caso do conector assíncrono, o problema derivado de um receptor ocioso é apenas o não recebimento das mensagens. O quão mais raro for para o receptor a solicitação de recebimento de uma mensagem, mais atrasado o recebimento estará em relação ao envio. No caso do conector síncrono, o problema derivado de um receptor ocioso é a paralisação do funcionamento do remetente. Como em sistemas síncronos existe a necessidade de coerência dos dados, o remetente não pode prosseguir com seu processamento até ter certeza de que o mesmo processamento é feito no sistema receptor.

Em relação ao envio de mensagens, o conector assíncrono é intrinsecamente mais eficiente, por não aguardar mensagem de reconhecimento por parte do receptor. Isso pode ser observado a partir do fato de que existem passos obrigatórios no fluxo do conector síncrono que não existem no fluxo do conector assíncrono, não sendo o oposto verdade.

O conector síncrono é assegurado do recebimento da mensagem por parte do receptor. Quando o recebimento não é confirmado, a parte remetente reenvia a mensagem. Esse mecanismo o torna mais confiável na maioria dos casos. Em caso de *timeout* no envio do *ack* acontece um falso negativo. O remetente reenvia a mensagem, apesar da mesma já ter sido recebida pelo receptor.

4 Avaliação dos Conectores

Este capítulo apresenta uma análise dos conectores propostos no contexto arquitetura de software, e mais especificamente no contexto de middlewares adaptativos. Conectores devem ser vistos como módulos responsáveis pela comunicação entre componentes utilizados em middlewares adaptativos. A meta da comparação desses módulos é associar desempenho e confiabilidade às características desses conectores. Esse conhecimento é necessário para o desenvolvedor encarregado de desenvolver o mecanismo de seleção dos conectores [4]. Esse mecanismo é o que dá o caráter adaptativo ao middleware. A informação utilizada para a seleção de um conector considera a atuação do mesmo no cenário em questão. Essa atuação leva em conta diversos aspectos como vazão, confiabilidade e assim por diante. Nós definimos essas métricas mais precisamente nesta seção, analisando as funcionalidades de um conector e os possíveis resultados de sua atuação.

Os conectores são uma forma de fornecer comunicação entre pares através de um acoplamento fraco (genérico). Os pares, no caso de um middleware adaptativo, são seus componentes. Essa comunicação é, em sua versão mais simplificada, a transmissão de bytes de uma ponta a outra. A transmissão ocorre em duas etapas externas ao conector: a solicitação de envio pelo componente remetente seguida pela solicitação de recebimento pelo componente receptor. A transmissão é bem-sucedida quando a mensagem enviada é recebida na outra ponta e ambos os componentes seguem seus fluxos normalmente. A transmissão é mal-sucedida quando a mensagem enviada não é recebida na outra ponta ou algum dos componentes fica em um estado de interrupção. Um dos motivos para uma transmissão mal-sucedida é a ausência de uma solicitação de recebimento após a solicitação de envio. Outros motivos para uma transmissão mal-sucedida são internos ao conector, e dependem da natureza das implementações. As métricas escolhidas devem capturar a presença desses cenários em cada modelo. Além disso, elas devem também possibilitar análises sobre o desempenho em cenários de transmissões bem-sucedidas.

4.1 Objetivos

Para a avaliação de nosso conector, as métricas a serem utilizadas devem ser úteis na avaliação da *confiabilidade* em situações de estresse (carga máxima). A métrica específica para medir confiabilidade é a taxa de sucesso. A taxa de sucesso deve medir a quantidade de mensagens que chegam ao seu destinatário em relação ao número de mensagens que foram enviadas. A confiabilidade de um conector deve variar entre cenários de baixa carga e sobrecarga, e os experimentos que avaliam taxa de sucesso devem envolver ambos os cenários, especialmente os de sobrecarga.

A qualidade de um conector não passa apenas pelo sucesso ou não da transmissão, e o desempenho também deve ser considerado. O desempenho depende de aspectos como a quantidade de tempo necessária para que uma mensagem seja enviada (latência), ou a quantidade de mensagens que podem ser enviadas em um determinado intervalo de tempo (vazão). Ambas as medidas medem o mesmo quesito desempenho de um conector e a diferença está em qual variável será fixada no experimento. No experimento em que se compara a latência, é necessário que o número de mensagens a ser enviada seja fixado. No experimento em que se compara a vazão, é necessário que o intervalo para o envio de mensagens seja fixado. Nossa opção foi pela análise baseada na vazão, por possuir uma métrica com um dimensional mais

explícito.

Para avaliar o conector síncrono, é necessária a definição de parâmetros internos que comandam o funcionamento do conector. Ao enviar uma mensagem, a parte remetente deve aguardar o recebimento da mensagem de reconhecimento. Caso o remetente não receba essa resposta, a mensagem original deve ser reenviada. O intervalo de tempo entre uma tentativa de envio e outra é fixo para um mesmo conector, na implementação descrita na Seção 3.5. Se o valor desse intervalo for muito pequeno, o remetente sempre reenviará a mensagem sem que haja necessidade. Se o valor for muito alto, o custo de uma falha de comunicação também se tornará muito alto. O valor escolhido representa um *trade-off* entre o número de duplicatas que chegam ao receptor e a vazão do conector, com os objetivos sendo a redução do número de duplicatas e maximização da vazão.

4.2 Experimentos

Nesta seção estão descritos os experimentos executados. Cada subseção define em detalhes a métrica a ser utilizada para avaliar desempenho, os parâmetros utilizados no experimento, as técnicas utilizadas para avaliar essa métrica, a carga de trabalho que será dada ao conector, os detalhes do experimento que mede o desempenho através de sua métrica, a análise e interpretação dos dados.

A Subseção 4.2.1 mede a vazão e o número de duplicatas enviadas de conectores síncronos com diferentes intervalos de espera entre reenvios de uma mensagem. Em todos os conectores síncronos deste estudo, o intervalo entre reenvios foi fixado para cada conector, para simplificar os modelos a serem analisados. Os conectores síncronos de melhor desempenho neste experimento serão comparados com os conectores assíncronos nos experimentos subsequentes.

A Subseção 4.2.2 avalia e compara a taxa de sucesso de todos os conectores. O experimento deve variar o número de solicitações para um conector de cada modelo, e comparar a taxa de sucesso entre os modelos. A taxa de sucesso pode ser medida comparando o número de mensagens recebidas ao número de mensagens enviadas. Neste experimento, vamos poder analisar características como escalabilidade do conector. Destacamos a influência de fatores como concorrência e sobrecarga em cada um dos modelos de conectores, e quais aqueles que apresentam maior escalabilidade. O melhor desempenho neste experimento representa o conector mais confiável e propício para um cenário em que o custo de uma falha de transmissão é alto.

A Subseção 4.2.3 avalia e compara a vazão de todos os conectores. Os testes deste experimento avaliam a vazão média dos modelos ao longo de um intervalo de tempo fixo. A vazão pode ser obtida ao se dividir o número total de mensagens recebidas pelo tempo intervalo pré-fixado. Este experimento permite a análise de características como eficiência e custo computacional.

A Subseção 4.2.4 analisa desempenho em cenários extremos para os melhores conectores assíncrono e síncrono. Os testes deste experimento tentam analisar se é possível ocorrer uma sobrecarga em relação a número de conectores, ao número de mensagens, ou a partir de uma combinação dos dois.

Todos os experimentos foram executados em uma máquina com processador Intel(R) Core(TM) i5-5200U com dois núcleos de frequência de 2.20GHz, 8 GB de memória e sistema operacional Windows 10 Home. Os experimentos foram executados utilizando o IDE Eclipse Neon 4.6.0. Para os experimentos com os conectores RabbitMQ, foi utilizada a versão

do RabbitMQ Server 3.7.5 e o cliente AMQP 4.0.2. Para os experimentos com o JeroMQ, foi utilizada versão 0.4.3 do projeto.

4.2.1 Tempo de espera do conector síncrono

No fluxo do conector síncrono, o remetente conclui o envio após receber uma mensagem de reconhecimento do receptor. Quando o remetente não recebe essa mensagem, ele assume que houve falha no envio da mensagem e a envia novamente. O tempo necessário para uma mensagem de reconhecimento do receptor chegar ao remetente após o envio da mensagem original pelo remetente é a latência do remetente de um conector síncrono. Se o tempo de espera pelo recebimento da mensagem de reconhecimento for inferior à latência do remetente do conector síncrono, o remetente sempre enviará múltiplas mensagens, independente da presença de falhas na comunicação. Caso o tempo de espera do remetente seja muito maior do que a latência do conector, o mesmo ficará desnecessariamente ocioso. Dessa forma, existe um valor ótimo para o tempo médio de espera. Esse valor depende da latência do remetente do conector 2, que por sua vez depende da latência entre receptor e conector com o serviço. Diferentes versões de conectores síncronos podem ser obtidas com variação do tempo de espera.

O experimento desta subseção tem o propósito de avaliar conectores síncronos baseados em RabbitMQ e JeroMQ que variam em seus tempos de espera para reenvio. O experimento se limita à análise da transmissão de mensagens pelo conector, que podem ser bem-sucedidas ou não. As métricas utilizadas nessa avaliação serão as *taxas de reenvio* e a *vazão* de cada um dos conectores. A *taxa de reenvio* é a quantidade de mensagens enviadas que já foram enviadas previamente em relação ao número total de mensagens enviadas. A *vazão* de um conector é a quantidade de mensagens enviadas dividida pelo intervalo de tempo necessário para o seu envio.

Além de variar o tempo de espera para reenvio, também variamos o número de mensagens do experimento. A variação no número total de mensagens enviadas será útil na análise da escalabilidade de um conector. Caso o conector demonstre uma diminuição em sua vazão ou um aumento nas taxas de reenvio será possível apontar uma menor escalabilidade por parte daquele conector.

Os experimentos foram feitos com os conectores em um cenário em que o remetente, o receptor e o serviço de filas funcionam na mesma máquina e estão localmente conectados. Para cada escolha de tempo de espera, um tempo total diferente é obtido para o envio do mesmo número total de mensagens. À medida que o tempo de espera se torna muito mais elevado que a latência do remetente do conector, o tempo necessário para o envio de todas as mensagens se aproxima ao tempo total de espera no envio dessas mensagens. No cenário em que o tempo de espera é inferior ao da latência mínima, o remetente nunca recebe a mensagem de reconhecimento a tempo, e o remetente sempre reenvia a mensagem pelo menos uma vez.

A latência média do remetente é variável em relação à probabilidade de uma mensagem ser transmitida com sucesso em uma tentativa e em relação ao tempo de espera do conector, e pode ser modelada da seguinte maneira:

$$T_M = \sum_{i=1}^{\infty} iT_E p(i)(\prod_{j=1}^{i-1} 1 - p(j)), onde: \quad (1)$$

- T_E é o tempo de espera para uma nova tentativa de envio da mensagem;

- $p(x)$ é a probabilidade da mensagem ser recebida x vezes T_E após ter sido enviada.

Porém a probabilidade de uma mensagem ser transmitida com sucesso em uma tentativa também é variável em relação ao tempo de espera do conector. Aproximando a densidade de probabilidade para latência de uma transmissão a uma distribuição normal, obtemos que a chance de uma transmissão ter sido bem sucedida até o tempo de espera especificado é dada por:

$$p(n) = \frac{p_k}{2}(1 + \frac{2}{\pi}\int_0^{\frac{nT_E-\mu}{\sigma\sqrt{2}}} e^{-t^2}dt) \tag{2}$$

- p_k é a probabilidade de não haver falha num único envio de uma mensagem;
- n é a quantidade de tempos de espera (T_E) que se passaram desde a tentativa em questão;
- μ é o tempo médio para que a mensagem de um envio bem-sucedido chegue ao seu destino;
- σ é o desvio padrão do tempo necessário para que a mensagem de um envio bem-sucedido chegue ao seu destino.

Para valores muito baixos de T_E, a probabilidade de sucesso $p(1)$ tende a ser nula, o que faz a latência média do remetente crescer. Para valores muito altos de T_E, $p(1)$ tende a p_k, que é constante, e T_M passa a crescer quase que linearmente com T_E. Um aumento no valor de $p(1)$ significa uma diminuição no valor de reenvios. Porém, para aumentar $p(1)$ é necessário aumentar T_E, que causa um aumento em T_M. T_M é o inverso da vazão, que desejamos maximizar.

Foram realizados testes com conectores cujos tempos de espera do conector eram de 100 microsegundos, 250 microsegundos, 500 microsegundos, 750 microsegundos, 1 milisegundo, 1 milisegundo e meio. 100 microsegundos e 1 milisegundo e meio são respectiva a menor e a maior latência de uma mensagem enviada pelos conectores deste experimento. Os testes foram feitos com cargas de trabalho de 100, 1 mil e 10 mil mensagens enviadas através do mesmo conector. O número de mensagens 100 é a menor potência de 10 cuja variação da vazão é sempre inferior à metade do valor médio. Os outros valores foram escolhidos para analisar o desempenho ao se aumentar a carga de trabalho em 1 e 2 ordens de grandeza. Foram usados conectores síncronos baseados em biblioteca de mensageria assíncrona (JeroMQ), e conectores síncronos baseados em um serviço de filas (RabbitMQ).

Para avaliar a vazão de cada conector, observamos o tempo necessário para o envio de todas as mensagens. A vazão representa o número de mensagens transmitidas dividido pelo número de segundos necessários para a conclusão do experimento. No caso de falha no envio de alguma mensagem, o número de mensagens utilizado no cálculo da vazão se refere ao número de envios bem-sucedidos.

O cálculo da taxa de reenvio é baseado na contagem de repetições de envio em relação ao número de envios bem sucedidos. O valor da taxa de reenvio é maior ou igual a zero, sem um limite superior. Quando a taxa de reenvio de um conector é baixa, isso significa que a primeira tentativa de envio da mensagem está sendo bem-sucedida, no geral.

A última métrica a ser observada é a taxa de sucesso de um conector. A taxa de sucesso de um conector corresponde ao número de mensagens que chegaram ao seu remetente em

relação à carga total de mensagens do experimento. O conector síncrono é projetado para que sua confiabilidade seja 100%, mas problemas com uso de memória, canais de comunicação ou sequência de falhas de comunicação podem ocasionar em uma mensagem não sendo transmitida com sucesso. Nos conectores utilizados neste experimento, o número limite era de 10 tentativas de envio de uma mensagem. A escolha por este número máximo de tentativas para o conector síncrono é por ele ser exatamente uma ordem de grandeza acima do número de tentativas do conector assíncrono.

Tabela 4.1: Vazão média (em mensagens por segundo) no envio de 100, 1 mil e 10 mil mensagens variando com o tempo de espera do conector baseado em RabbitMQ.

Tempo de Espera do Conector (T_E)	0.25ms	0.5ms	0.75ms	1ms	1.5ms
Vazão para 100 mensagens	252.38	251.40	260.07	264.64	253.99
Vazão para 1 mil mensagens	329.56	333.22	333.42	398.75	329.25
Vazão para 10 mil mensagens	379.00	378.30	376.80	508.57	368.11

A Tabela 4.1 mostra a vazão média do conector síncrono baseado em RabbitMQ, variando em relação a sua carga total de trabalho e o valor de intervalo até o reenvio. Os melhores resultados concentraram-se na coluna do conector com tempo de espera de 1 milisegundo. O motivo para que um conector com tempo de espera maior obtenha melhor desempenho que conectores com menor tempo de espera vem do fato de que o RabbitMQ é quase sempre bem-sucedido na transmissão da mensagem, e o reenvio é desnecessário. Neste cenário, o reenvio significa uma competição por recursos computacionais que não são necessários à parte remetente do conector. A Tabela 4.2 apresenta as taxas de reenvio dos mesmos conectores.

Tabela 4.2: Taxa de reenvio no envio de 100, 1 mil e 10 mil mensagens variando com o tempo de espera do conector baseado em RabbitMQ.

Tempo de Espera do Conector (T_E)	0.25ms	0.5ms	0.75ms	1ms	1.5ms
Taxa de reenvio para 100 mensagens	0%	0%	0%	1.00%	0.45%
Taxa de reenvio para 1 mil mensagens	0.69%	0.58%	0.43%	0.39%	0.55%
Taxa de reenvio para 10 mil mensagens	0.08%	0.09%	0.13%	0.08%	0.13%

O dado sobre as taxas de reenvio em conectores baseados em RabbitMQ comprova a eficiência do serviço de filas para a transmissão de mensagens. Deve ser observado que embora uma alta taxa de reenvio indique falhas de comunicação, reenvios podem acontecer também por atrasos. Valores tão próximos de zero como os da Tabela 4.2 são mais provavelmente derivados de demora no desenfileiramento do que de falhas de comunicação.

Em todos os testes feitos neste experimento, 100% das mensagens enviadas chegaram ao componente receptor ligado ao conector baseado em RabbitMQ. Isso é indicativo de escalabilidade do conector, que não foi afetado por uma demanda alta de mensagens. Outro indicativo da escalabilidade do conector é o fato de que a sua vazão não diminuiu com o aumento da carga de trabalho.

Observando a Tabela 4.3, é mais difícil de apontar uma tendência da vazão do conector

Tabela 4.3: Vazão média (em mensagens por segundo) no envio de 100, 1 mil e 10 mil mensagens variando com o tempo de espera do conector baseado em JeroMQ.

Tempo de Espera do Conector (T_E)	0.25ms	0.5ms	0.75ms	1ms	1.5ms
Vazão para 100 mensagens	72.14	63.72	78.11	80.63	81.53
Vazão para 1 mil mensagens	117.81	116.66	121.01	121.40	119.47
Vazão para 10 mil mensagens	69.01	68.01	67.13	66.67	65.20

baseado em JeroMQ que em relação ao seu tempo de espera. Em relação ao conector baseado em RabbitMQ, o conector baseado em JeroMQ possui uma vazão menor em todos os cenários. Uma vazão menor com tempos similares de espera é indicativo de que as outras operações do conector são mais custosas. Apesar de não haver um candidato óbvio, o conector com tempo de espera de 1 milisegundo tem a maior vazão nos cenários de 100 e 1000 mensagens e uma vazão similar a todos os outros no cenário de 10 mil mensagens.

Tabela 4.4: Taxa de reenvio no envio de 100, 1 mil e 10 mil mensagens variando com o tempo de espera do conector baseado em JeroMQ.

Tempo de Espera do Conector (T_E)	0.25ms	0.5ms	0.75ms	1ms	1.5ms
Taxa de reenvio para 100 mensagens	3.19%	3.07%	2.45%	2.83%	2.72%
Taxa de reenvio para 1 mil mensagens	0.44%	0.24%	0.29%	0.38%	0.32%
Taxa de reenvio para 10 mil mensagens	12.92%	11.98%	11.08%	10.83%	10.74%

Na Tabela 4.4, observamos um súbito aumento na taxa de reenvio de mensagens no conector baseado em JeroMQ no cenário do envio de mais mensagens. Esse aumento surge a partir de um crescimento no número de falhas de comunicações causado por um envio de muitas mensagens em um curto intervalo de tempo. Em todos os testes executados neste experimento, apenas no cenário de envio de 10 mil mensagens pelo conector baseado em JeroMQ ocorreram casos de mensagens que nunca chegaram ao receptor. A quantidade de falhas por teste não estava relacionada ao tempo de espera do conector, mas à quantidade de mensagens a serem enviadas, com uma média de 11.46 mensagens que nunca chegaram ao receptor de cada 10 mil. Os outros casos tiveram 100% de sucesso na transmissão.

4.2.2 Confiabilidade

A confiabilidade é um requisito não funcional que pode ser mensurado através da taxa de sucesso. A taxa de sucesso um conector é a probabilidade de uma mensagem ser transmitida pelo mesmo com sucessso. No conector assíncrono, apenas uma tentativa é feita, e o seu resultado é desconhecido ao remetente. Quando uma tentativa falha no conector assíncrono, o conector seguirá assumindo que o receptor recebeu a mensagem, e o receptor nunca estará ciente do envio da mensagem. Esse funcionamento é caracterizado como melhor esforço [28]. O receptor síncrono repete as tentativas de transmissão, enquanto a mesma não é concluída, mas também pode falhar em alguns cenários. As possibilidades de falha na transmissão de

ambos os tipos de conectores são variadas, e incluem falhas de comunicação, indisponibilidade de recursos computacionais e receptor ocioso.

No conector síncrono, um número pré-determinado de tentativas é feito até que se desista da transmissão da mensagem. Para o conector síncrono, o fator mais importante é a igualdade das informações presentes em ambos os nós. Na Subseção 4.2.1 está definida a probabilidade de uma tentativa de transmissão ser bem sucedida em função do seu tempo de espera. Dado que fixamos o tempo de espera do conector síncrono deste estudo, a probabilidade de sucesso de uma transmissão é definida em função de quantas tentativas foram feitas antes da mesma e do tempo passado desde o envio de cada mensagem 2. Podemos aproximar a probabilidade de sucesso de uma transmissão (com repetição do envio em caso de falha) como:

$$P_n = \sum_{i=1}^{n} p_i (\prod_{j=1}^{n-1} 1 - p_j) \tag{3}$$

Essa função é crescente com n e com p, que por sua vez é crescente com o tempo de espera. Dessa forma, espera-se que um conector síncrono com maior tempo de espera e maior número máximo de tentativas é mais confiável.

Os modelos avaliados neste experimento são os conectores síncronos e assíncronos baseados em serviço de filas (RabbitMQ) e os conectores síncronos e assíncronos baseados em biblioteca de mensageria assíncrona (JeroMQ). Ambos os conectores síncronos tiveram seus intervalos de espera até reenvio fixados em 1 milisegundo, e seu número máximo de tentativas de reenvio fixado em 10. O tempo de espera para reenvio escolhido é o do conector de melhor desempenho do experimento anterior. O número máximo de tentativas do conector síncrono é o de exatamente uma ordem de grandeza acima do número de tentativas do conector assíncrono. Cada conector foi testado em três diferentes cenários: transmissão de 100 mensagens, 1 mil mensagens e 10 mil mensagens. O número de mensagens 100 é a menor potência de 10 cuja variação da vazão é sempre inferior à metade do valor médio. Os outros valores foram escolhidos para analisar o desempenho ao se aumentar a carga de trabalho em 1 e 2 ordens de grandeza. O único dado observado neste experimento foi o número de transmissões bem-sucedidas. A confiabilidade é calculada a partir da divisão de transmissões bem-sucedidas pelo número total de mensagens enviadas.

O experimento consiste da tentativa de transmissão serializada de todas as mensagens do cenário de testes atual. O número de transmissões bem sucedidas é utilizado para calcular a taxa de sucesso do conector no teste. Os resultados do experimento podem ser observados na tabela 4.5.

Tabela 4.5: Taxa de sucesso no envio de 100, 1 mil e 10 mil mensagens com cada modelo de conector.

Modelo do Conector	RabbitMQ		JeroMQ	
	Assíncrono	Síncrono	Assíncrono	Síncrono
Taxa de sucesso para 100 mensagens	98.40%	100%	92.62%	100%
Taxa de sucesso para 1 mil mensagens	99.72%	100%	98.44%	100%
Taxa de sucesso para 10 mil mensagens	99.94%	100%	97.03%	99.90%

O resultado do experimento demonstra uma confiabilidade mais alta nos conectores baseados em RabbitMQ. O resultado do teste com o conector baseado em RabbitMQ síncrono foi o melhor possível, o que já tinha sido observado no experimento da subseção anterior. O conector assíncrono baseado em RabbitMQ obteve um resultado muito bom, mas ainda ocorrem falhas nas transmissão de algumas mensagens. O conector síncrono JeroMQ performa melhor que o conector assíncrono RabbitMQ, mas consegue o melhor resultado apenas para as duas menores cargas de mensagens. O pior desempenho é claramente o do conector assíncrono JeroMQ, já que aproximadamente 3% de suas mensagens não são transmitidas com sucesso para uma carga grande de mensagens. Seu desempenho também é pior nos outros cenários.

4.2.3 Vazão

A vazão de um conector é a sua velocidade de transmissão de mensagens. Essa capacidade pode ser analisada pelo ponto de vista do receptor, que analisa quantas mensagens foram recebidas em um determinado intervalo, ou quanto tempo é necessário para o envio de um determinado número de mensagens. Essa métrica não avalia o sucesso no envio das mensagens, e pode apontar um valor que ainda precisa ser associado à confiabilidade do conector para que uma conclusão possa ser tirada em relação ao mesmo. Porém, em cenários onde a probabilidade de sucesso na transmissão de uma mensagem tende a 1, como ficou evidente que é o caso de alguns conectores deste estudo, a velocidade do conector é a métrica que melhor define seu desempenho geral.

No conector síncrono, a conclusão do envio de uma mensagem é posterior ao seu recebimento. Dessa forma, no envio de um grande número de mensagens, o início do envio de uma mensagem é sempre posterior à conclusão do recebimento da mensagem anterior. A vazão do conector diz respeito ao intervalo de tempo médio de uma transmissão, para o conector síncrono. No conector síncrono, o envio é concluido antes do recebimento iniciar. Isso torna possível que várias mensagens sejam enviadas antes que a primeira seja recebida. Por esse paralelismo, diferentemente do conector síncrono, não há uma relação entre o tempo médio de uma transmissão e a vazão do conector.

No experimento da Subseção 4.2.1, analisamos a vazão de conectores síncronos como uma forma de decidir o melhor tempo de espera para cada modelo. O objetivo do experimento era comparar conectores baseados na mesma tecnologia, mas com tempo de espera diferente. Este objetivo foi alcançado, mas também foi possível comparar os conectores síncronos baseados em tecnologias diferentes. No primeiro experimento desta subseção também compararemos a vazão dos conectores, mas analisaremos todos os 4 modelos.

Os modelos de conectores a serem avaliados neste experimento são os mesmos modelos da Subseção 4.2.2. Desta vez, eles serão avaliados apenas pela métrica da vazão. Os modelos são os conectores baseados em RabbitMQ síncrono e assíncrono, e os conectores baseados em JeroMQ síncrono e assíncrono.

Para avaliar a vazão de cada conector, observamos a quantidade de mensagens que podem ser transmitidas com sucesso pelo conector num intervalo de 1 minuto. A vazão média, em mensagens por segundo, é calculada dividindo o número total de mensagens transmitidas por 60. Também foi observado o tempo após a conclusão da transmissão de cada mensagem individual, para que pudéssemos analisar o número de mensagens transmitidas ao longo do tempo.

A primeira forma de avaliação, presente na Tabela 4.6, é indicativo de um desempenho

superior dos conectores baseados em RabbitMQ. Mesmo o conector baseado em RabbitMQ síncrono, cuja taxa de sucesso foi 100% em todos os cenários do experimento de confiabilidade, possui uma vazão maior que a do conector baseado em JeroMQ assíncrono. Já em relação ao conector baseado em JeroMQ síncrono, foi demonstrado que, como no primeiro experimento, sua vazão é inferior ao conector baseado em RabbitMQ síncrono.

Tabela 4.6: Vazão média (em mensagens por segundo) de cada modelo de conector após 1 minuto.

Modelo do Conector	RabbitMQ		JeroMQ	
	Assíncrono	Síncrono	Assíncrono	Síncrono
Vazão média após 1 minuto	1571.32	546.18	271.12	103.23

A vazão dos conectores síncronos é mais que duas vezes superior à da sua contraparte síncrona. Isso é esperado, dado que o fluxo do conector síncrono envolve duas mensagens e o do conector assíncrono apenas contém uma mensagem. Além disso, é possível transmitir uma nova mensagem no conector assíncrono antes que a última seja recebida, o que permite o paralelismo em seu fluxo.

A Tabela 4.6 contém apenas a vazão média final do experimento. Para analisar a vazão de cada conector ao longo do experimento devemos olhar para o gráfico da Figura 4.1, que contém a relação entre o número de transmissões concluídas em relação ao tempo. A vazão de cada momento pode ser obtida a partir do coeficiente angular do gráfico no instante específico.

No gráfico da Figura 4.1, a ordem dos conectores em função de sua vazão é a mesma ao longo de todo o experimento. A vazão do conector RabbitMQ assíncrono varia nos primeiros 6 segundos, mas logo a mesma se torna constante, e a sua curva se aproxima de uma reta. O conector RabbitMQ síncrono tem vazão constante durante todo o experimento. O conector JeroMQ assíncrono inicia com vazão constante e próxima ao do RabbitMQ síncrono, mas por volta dos 42 segundos atinge um estado de sobrecarga e sua vazão assume um novo valor constante e muito menor. O conector JeroMQ síncrono também tem vazão inicial constante e atinge um estado de sobrecarga.

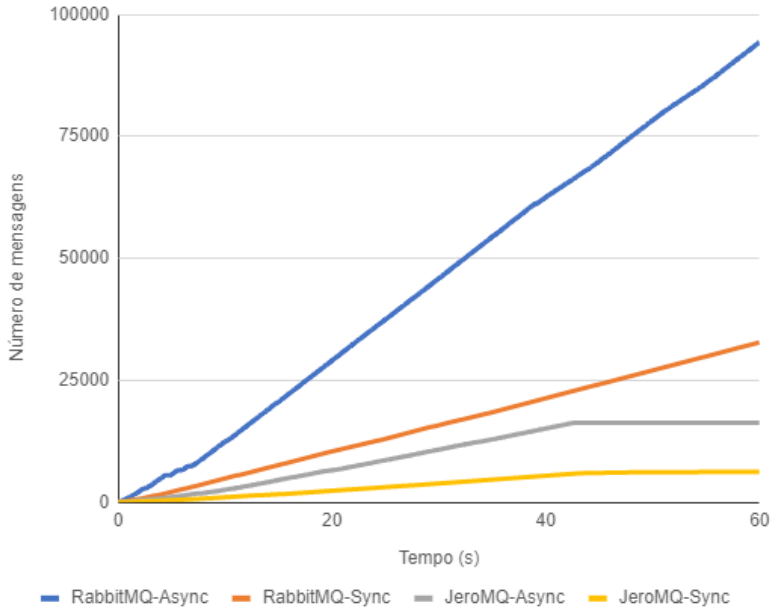

Figura 4.1: Número de mensagens enviadas ao longo do tempo através de conector baseado em RabbitMQ assíncrono, conector baseado em RabbitMQ síncrono, conector baseado em JeroMQ assíncrono, conector baseado em JeroMQ síncrono.

Analisando a tabela e o gráfico deste experimento, podemos apontar que ambos os conectores RabbitMQ performam melhor que suas versões JeroMQ. Essa conclusão também pode ser obtida a partir da análise dos outros experimentos. Apesar de se utilizar de uma forma de comunicação direta, o JeroMQ oferece uma solução menos escalável e eficiente que o RabbitMQ para o desenvolvimento de conectores.

4.2.4 Escalabilidade

Apesar de uma vazão mais alta, a escolha pelo conector RabbitMQ assíncrono nem sempre pode ser feita. Os requisitos do sistema devem definir se a escolha a ser feita deve ser por um conector síncrono ou por um conector assíncrono. Os requisitos a serem analisados são sincronia e confiabilidade. Em um sistema no qual há necessidade de sincronia entre componentes, a opção pelo conector assíncrono não pode ser feita. Se a confiabilidade for um requisito muito mais importante que desempenho, a escolha pode ser pelo conector de menor vazão, mas com confiabilidade ligeiramente mais alta. Isso aconteceria num cenário onde o custo de uma mensagem não transmitida seria maior que o custo de uma latência mais alta. Por isso, não é possível tirar conclusões imediatas sobre qual conector baseado em RabbitMQ obteve o melhor resultado sem que se saiba qual o sistema a ser utilizado. Ambos os conectores são possíveis melhores escolhas para diferentes sistemas.

Os conectores baseados em RabbitMQ apresentaram os melhores desempenhos nas métricas

deste estudo. Uma preocupação ao se utilizar um módulo como um conector é o limite de suas capacidades. Nos experimentos das subseções anteriores, foram identificadas quedas de desempenho dos conectores baseados em JeroMQ sob uma carga grande de mensagens. Para identificar se o mesmo acontece com os conectores RabbitMQ, estendemos as cargas máximas de mensagens e analisamos a vazão em tais cenários.

Neste experimento, dois fatores foram selecionados. Como feito anteriormente, o número de mensagens variou até 10 mil mensagens. O outro fator foi o número de conectores utilizados pela componente remetente, que variou neste experimento de 1 a 100. No cenário mais extremo, a componente remetente envia 10 mil mensagens através de cada um de seus 100 conectores, e a vazão medida se refere ao total de mensagens enviadas pelo componente remetente transmitidas com sucesso.

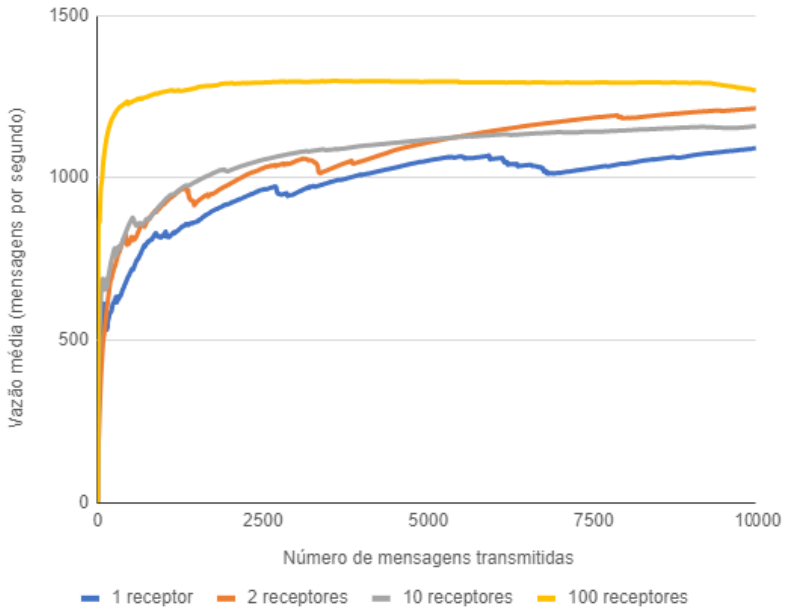

Figura 4.2: Gráfico da vazão média do remetente em função do número de mensagens transmitidas através do conector baseado em RabbitMQ assíncrono para diferentes números de receptores.

A Figura 4.2 contém a vazão média dos conectores RabbitMQ assíncrono nos testes deste experimento em função do número de mensagens transmitidas até o momento. O grau de variação da vazão entre diferentes cenários do experimento é baixo. A vazão oscila entre 1000 e 1400 mensagens por segundo na maior parte dos cenários. Essa variação não segue uma tendência, e é causada em sua maioria por fatores que afetam o experimento que não podem ser controlados. O desempenho dos conectores não é afetada por sobrecarga de número de mensagens ou de conectores para o modelo assíncrono.

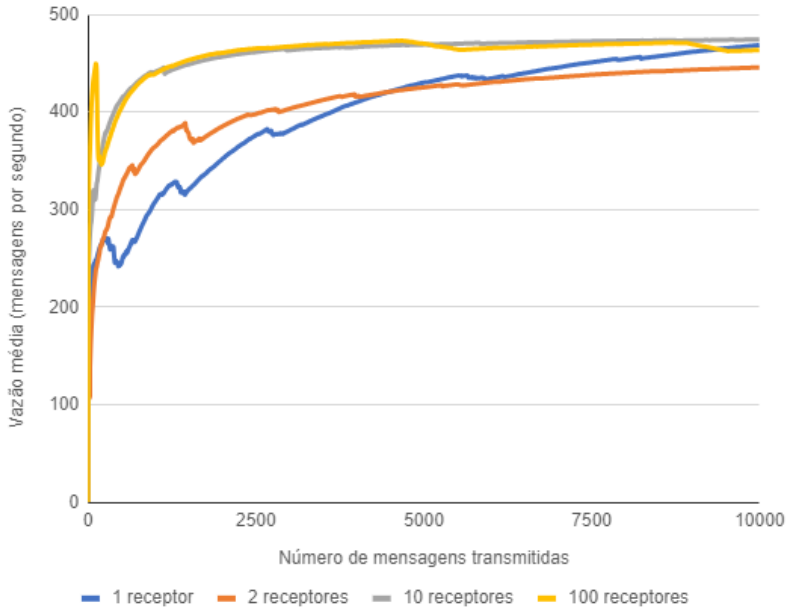

Figura 4.3: Gráfico da vazão média do remetente em função do número de mensagens transmitidas através do conector baseado em RabbitMQ síncrono para diferentes números de receptores.

Assim como no caso do conector assíncrono, o conector RabbitMQ síncrono não sofre grande variações em sua vazão, mesmo nos cenários extremos, como pode ser visto na Figura 4.3. A taxa de sucesso do conector também se manteve, uma vez que todas as mensagens em todos os testes com o conector síncrono chegaram ao seu destino.

5 Conclusão e Trabalhos Futuros

Nós analisamos 4 tipos de conectores neste estudo: conector baseado em RabbitMQ síncrono, conector baseado em RabbitMQ assíncrono, conector baseado em JeroMQ síncrono e conector baseado em JeroMQ assíncrono. Eles foram comparados em relação à tecnologia utilizada para a comunicação interna do conector e em relação à presença ou não de mecanismo de sincronia.

As métricas analisadas foram a taxa de sucesso do conector, sua vazão, e, para o conector síncrono, a taxa de reenvio. Os conectores eleitos como melhores a partir dos dados dos experimentos também tiveram sua escalabilidade testada. A conclusão descreve os resultados obtidos e os cenários onde a utilização de cada um deles seria mais apropriada.

5.1 Conclusões

O propósito primário deste estudo era analisar duas tecnologias como base para a comunicação de um conector. Os dois candidatos foram o serviço de filas RabbitMQ e a biblioteca de mensageria assíncrona JeroMQ, uma implementação em Java da biblioteca ZeroMQ. Em todos os experimentos que serviram de base para comparação entre os conectores RabbitMQ e os conectores JeroMQ, o RabbitMQ levou vantagem nas métricas adotadas. A taxa de sucesso dos conectores RabbitMQ foi a máxima, ou muito próxima da máxima, nos experimentos realizados. A confiabilidade dos conectores JeroMQ era reduzida em cenários com maior carga operacional, sendo consistentemente inferior à confiabilidade dos conectores RabbitMQ. O mesmo aconteceu com a vazão dos conectores analisados. Em cenários de menor carga de trabalho, os conectores JeroMQ funcionavam como esperado, apesar de performarem abaixo dos conectores RabbitMQ. Em cenários com maior carga de trabalho, os conectores JeroMQ ficaram sobrecarregados, e sua vazão foi prejudicada. Os conectores RabbitMQ, por sua vez, não tiveram sua vazão afetada por uma carga maior. A seleção pelo melhor conector síncrono e assíncrono foi pelo conector RabbitMQ em ambos os casos.

Na análise do nosso conector assíncrono, pudemos verificar que o RabbitMQ é um serviço de filas robusto e escalável. A rara ocorrência de falhas no serviço faz com que a confiabilidade do conector seja mais alta. Em cenários sem problemas de comunicação, sua taxa de sucesso tende ao valor máximo. A velocidade de envio do conector assíncrono é máxima, uma vez que o envio de uma mensagem é atômico. O limite da vazão do conector assíncrono foi superior a 1000 mensagens por segundo, o que significa que o conector não representaria um gargalo no sistema a não ser que houvesse uma demanda pelo envio de mais de 1000 mensagens a cada segundo. Em relação a sincronia, não há garantias no conector assíncrono, uma vez que não há mecanismos que informem à componente remetente que a transmissão foi bem sucedida e o componente receptor solicitou e recebeu a mensagem.

Um dos experimentos deste estudo tinha o propósito de encontrar o tempo de espera para reenvio ótimo para os conectores síncronos. O conector RabbitMQ síncrono obteve vazão máxima com o tempo de espera de 1 milisegundo, e a taxa de reenvio não foi diferente de conectores com tempo de espera diferente, sendo inferior a 1% em todos os cenários. A confiabilidade do conector RabbitMQ síncrono foi máxima em todos os experimentos realizados, dando garantias de recebimento da mensagem por parte do componente receptor. As medições apontaram que a vazão do conector síncrono é inferior à metade da vazão de sua contraparte síncrona. Também ficou demonstrado que a vazão não era afetada por um aumento na carga de trabalho para o conector. O limite da vazão do conector síncrono oscilou entre 300 e 500

mensagens por segundo, o que significa que o conector não é um gargalo em qualquer sistema que tenha uma média de requisições por segundo inferior a 300.

A opção pelo uso do conector síncrono e assíncrono deve ser feita baseada nos requisitos de sincronia e confiabilidade do sistema. O conector síncrono é uma melhor opção para sistemas críticos, onde a confiabilidade deve ser máxima. Outro cenário onde o conector síncrono é necessário é o aquele em que o componente remetente pode seguir com seu funcionamento apenas após o recebimento da mensagem por parte do destinatário.

5.2 Trabalhos futuros

A análise feita neste estudo deixa em aberto a possibilidade de novas análises sobre estes conectores e sobre conectores baseado neles.

Para o conector síncrono, cabe o estudo da possibilidade de um conector com características variáveis, como seu tempo de espera e número máximo de tentativas de reenvio. No conector síncrono deste estudo, o tempo de espera é uma característica do conector, mas existem possíveis vantagens de se utilizar um conector que tenha seu tempo de espera como uma variável. Também faz sentido que se analise a possibilidade de um conector seletivamente síncrono, para o qual apenas mensagens marcadas como tal devem ser transmitidas em uma rotina síncrona.

O cenário deste estudo é bastante restritivo, e pode não ser representativo de uma grande parcela dos casos de uso de um conector. A introdução de falhas de comunicação deve aproximar a avaliação dos conectores deste estudo à realidade destes casos de uso.

Também é interessante que se analise a possibilidade de outros serviços serem utilizados para a produção de conectores, como a implementação original da biblioteca de mensageria assíncrona ZeroMQ [5].

Finalmente, é importante que exista um algoritmo de decisão que faça a escolha entre qual conector deve ser usado pelo middleware em cada momento. Um estudo sobre os resultados deste algoritmo descritos junto aos cenários para os quais a escolha foi feita é necessário. Também é necessário analisar a influência das mudanças em desempenho do middleware após a troca de conectores.

Referências Bibliográficas

[1] What can rabbitmq do for you? Last Access: 2017-11-26.

[2] Bjarne Poulsen Bo Petersen, Henrik Bindner and Shi You. Smart grid communication middleware comparison. 2016.

[3] D Pakkala, J Peralla, and E Niemela. A component model for adaptive middleware services and applications. *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on Advanced Applications (EUROMICRO 2007)*, page 2, 2007.

[4] Nelson S Rosa. Building adaptive middleware using software architecture principles. page 1, 2016.

[5] Zeromq: Distributed messaging. Last Access: 26-10-2017.

[6] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SOFTWARE ENGINEERING NOTES*, 17.

[7] Jan Bosch and Peter Molin. Software architecture design: Evaluation and transformation.

[8] Jan Bosch. Software architecture: The next step. (1):194–199, 05 2004.

[9] Wermelinger Michel and Luiz Fiadeiro José. Connectors for mobile programs. *TRANSACTIONS ON SOFTWARE ENGINEERING*, 24(5):331–332, 1996.

[10] Oussalah Mourad, Smeda Adel, and Khammaci Tahar. An explicit definition of connectors for component-based software architecture. *IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, 11, 2004.

[11] AbdelHamid Amr and Zong Peng. A new software connector for distributed component simulation platforms. 2015.

[12] Garlan David and Spitznagel Bridget. A compositional approach for constructing connectors. page 148, 2001.

[13] Dashofy Eric M, Medvidovic Nenad, and N Taylor Richard. Using off-the-shelf middleware to implement connectors in distributed software architectures. *Proceedings of the 1999 International Conference on Software Engineering*, (3):2, 1999.

[14] Issarny Valérie, Kloukinas Christos, and Zarras Apostolos. Systematic aid for developing middleware architectures. *Communications of the ACM*, page 1, 2002.

[15] Morris Karl, Allison Mark, M. Costa Fábio, Wei Jinpeng, and Clarke Peter J. An adaptive middleware design to support the dynamic interpretation of domain-specific models. *Information and Software Technology*, 62(1):21–41, 2013.

[16] da Rocha Tarcisio and Maria Beatriz Felgar de Toledo. Managing complexity in open adaptive middleware. *IEEE International Conference on Computational Science and Engineering*, 11(1,2):200–201, 2008.

[17] Rosa Nelson. Middleware adaptation through process mining. *IEEE International Conference on Advanced Information Networking and Applications*, 31, 2017.

[18] Sun Jingtao and Duan Sisi. A self-adaptative middleware for efficient routing in distributed sensor networks. *IEEE International Conference on Systems, Man, and Cybernetics*, (II.B):322–323, 2015.

[19] Oliveira Vasconcelos Rafael, Talavera Luis, Vasconcelos Igor, and Roriz Marcos andEndler Markus. An adaptive middleware for opportunistic mobile sensing. *International Conference on Distributed Computing in Sensor Systems*, pages 1–3, 2015.

[20] Park Soojin and Song JaeSung. Self-adaptive middleware framework for internet of things. *Global Conference on Consumer Electronics (GCCE)*, 4(I-III):81–82, 2015.

[21] Carlo Ghezzi. Adaptive software needs continuous verification. pages 3–4, 09 2010.

[22] Glaucia M. M. Campos Nelson S. Rosa and David J. M. Cavalcanti. Lightweight formalisation of adaptive middleware. 2018.

[23] Valeriu Manuel Ionescu. The analysis of the performance of rabbitmq and activemq. 2015.

[24] Huang Chun-Che and Kusiak Andrew. Modularity in design of products and systems. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, 28:1, 1998.

[25] Nicolas Estrada and Hernan Astudillo. Comparing scalability of message queue system: Zeromq vs rabbitmq. *XLI Latin American Computing Conference (CLEI)*, 2015.

[26] Clients and developer tools. Last Access: 2017-11-26.

[27] Chen Jiwei, Lee Yeng Zhong, Gerla Mario, and Sanadidi M.Y. Tcp with delayed ack for wireless networks. *International Conference on Broadband Communications, Networks and Systems, 2006. BROADNETS 2006.*, 3(I, II):1–3, 2006.

[28] Comer D and Yavatkar R. Flows: performance guarantees in best effort delivery systems. *INFOCOM '89. Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies. Technology: Emerging or Converging, IEEE*, (3):102, 1989.