Uso de padrão AMQP para transporte de mensagens entre atores remotos

Thadeu de Russo e Carmo
thadeurc@ime.usp.br

Orientador: Prof. Dr. Francisco da Rocha Reverbel
Instituto de Matemática e Estatística
Universidade de São Paulo

Maio, 2012

Motivação

Explorar a potencial sinergia entre duas classes de sistemas de *software*:

1. Sistemas corporativos: composta de sistemas de *middleware* orientados a mensagens e *message brokers*
2. Programas concorrentes: composta pelas implementações do modelo de atores

Motivação – Sistemas corporativos

- *Middlewares* orientados a mensagem (MOMs) trabalham com troca assíncrona de mensagens
- Formam base para simplificar o desenvolvimento de sistemas
- Possuem suporte e mecanismos para gerenciamento robusto de erros e garantia de entrega de mensagens
- São frequentemente apresentados como tecnologia que pode mudar a maneira como sistemas distribuídos são construídos [Alonso et al 2004]

Motivação – Sistemas corporativos

- MOMs são um tanto quanto inflexíveis em relação a filtragem e roteamento de mensagens
- *Message brokers* são descendentes diretos dos MOMs e endereçam essas limitações

Motivação – Sistemas corporativos

Os protocolos usados por *message brokers* variam de produto para produto. Algumas abordagens existentes:

- A especificação Java *Message Service* (JMS) define uma API padrão para que programas Java possam interagir com *message brokers*
- O padrão *Advanced Message Queuing Protocol* (AMQP) é uma proposta recente de padronização de protocolo para *message brokers*

Motivação – Programas concorrentes

Processadores *multicore* impactam a maneira que programas são escritos:

- Programas precisam ser escritos de maneira concorrente para poderem usufruir dos ganhos de desempenho dos processadores *multicore* [Sutter 2005]
- Abordagem convencional com o uso de travas, além de complexa, é limitada e não permite a composição das travas de modo seguro [Jones 2007]
- A maioria das linguagens de programação não são adequadas para a criação de tais programas [Sutter & Larus 2005]

Motivação – Programas concorrentes

Modelos não convencionais de programação concorrente:

- *Software Transactional Memory* (STM) – mecanismo de controle análogo às transações de bancos de dados
- Atores – troca de mensagem assíncrona entre processos

Motivação

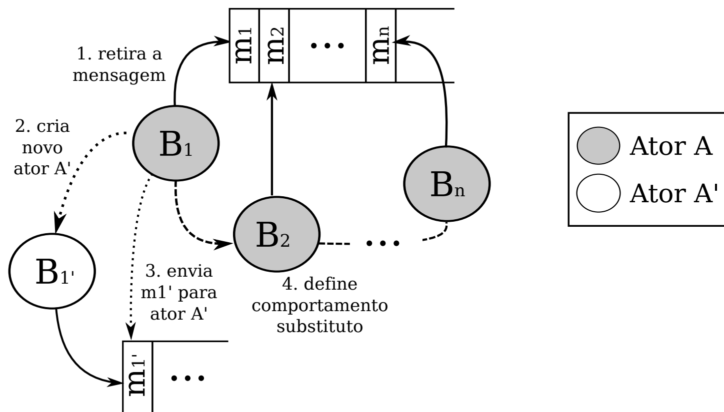
Potencial sinergia entre as duas classes:

- Atores em diferentes nós de uma rede de computadores podem trocar mensagens entre si
- *Message brokers* provêm robustez para a troca de mensagens entre entidades de diferentes nós de uma rede de computadores

Objetivos

- Criar de uma implementação em Scala do modelo de atores que use o padrão AMQP para o transporte de mensagens entre atores remotos
- Comparar o desempenho do protótipo desenvolvido neste trabalho com o da implementação original de atores do projeto Akka

Atores



Atores

Algumas implementações:

- Linguagens: Axum, SALSA e Erlang
- C++: Act++, Thal e Theron
- Smalltalk: Actalk
- Python: Parley e Stackless Python
- Ruby: Stage e Rubinius
- .Net: Asynchronous Agent Library e Retlang
- Java: Akka, Kilim, Jetlang e Actor Foundry
- Scala: Scala Actors, Akka e Scalaz

Atores no projeto Akka

O projeto Akka disponibiliza dois tipos de atores:

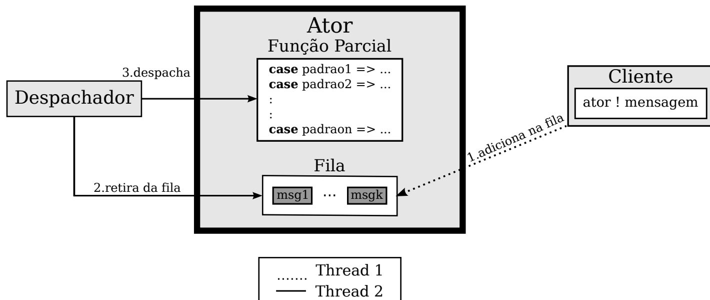
1. Atores locais: um ator local é um ator que pode receber mensagens apenas de atores residentes na mesma máquina virtual
2. Atores remotos: um ator remoto é um ator que pode receber mensagens de quaisquer outros atores, inclusive daqueles residentes em outras máquinas virtuais

Atores locais

Declaração e uso de um ator:

```
class SampleActor(val name: String) extends akka.actor.Actor {
  def this() = this("No name")

  def receive = {
    case "hello" => println("%s received hello".format(name))
    case _ => println("%s received unknown".format(name))
  }
}

/* theActor representa uma LocalActorRef */
val theActor = Actor.actorOf[SampleActor].start
theActor ! "hello"
```

Envios assíncronos e sem resposta – !



Envios síncronos com resposta – !! e !!!

Interface do método !!!:

```
def !!!(message: Any, timeout: Long = this.timeout) (implicit sender: Option[ActorRef] =
        None): Future[T] = {
...
}
```

Feição CompletableFuture:

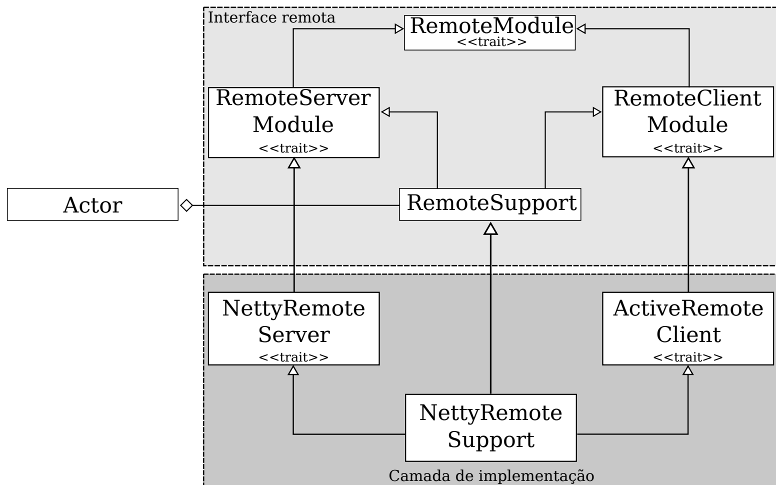
```
trait CompletableFuture[T] extends Future[T] {
  def completeWithResult(result: T)
  def completeWithException(exception: Throwable)
  def completeWith(other: Future[T])

  // declarados em akka.dispatch.Future
  def result: Option[T]
  def await : Future[T]
}
```

Envios síncronos com resposta – !! e !!!

Passos executados na execução do método !!!:

1. Uma instância de CompletableFuture é criada
2. A mensagem é depositada na fila do ator junto com uma referência para a instância do passo 1
3. Uma outra referência para a instância do passo 1 é devolvida a quem fez a chamada do método
4. O resultado do processamento da mensagem é utilizado para completar a instância criada no passo 1

Atores remotos



Atores remotos

Registro de um ator remotamente acessível:

```
object SampleRemoteServer extends Application {
  Actor.remote.start("localhost", 2552)
  Actor.remote.register("hello-service", Actor.actorOf[SampleActor])
}
```

Pesquisa e uso de um ator remotamente acessível:

```
object SampleRemoteClient extends Application{
  /* helloActor representa uma RemoteActorRef */
  val helloActor = Actor.remote.actorFor("hello-service", "localhost", 2552)
  helloActor ! "Hello"
}
```

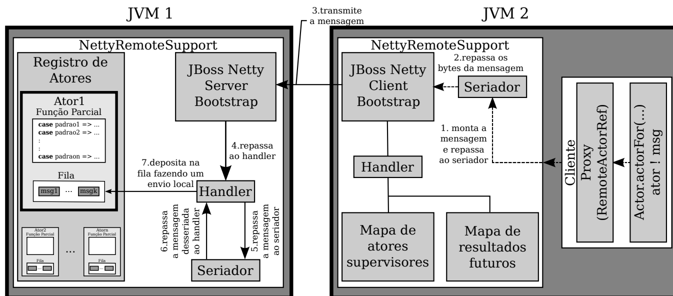
Formato e seriação das mensagens

```
message RemoteMessageProtocol {
  required UuidProtocol uuid = 1;
  required ActorInfoProtocol actorInfo = 2;
  required bool oneWay = 3;
  optional MessageProtocol message = 4; // <- payload da mensagem
  optional ExceptionProtocol exception = 5;
  optional UuidProtocol supervisorUuid = 6;
  optional RemoteActorRefProtocol sender = 7;
  repeated MetadataEntryProtocol metadata = 8;
  optional string cookie = 9;
}
```

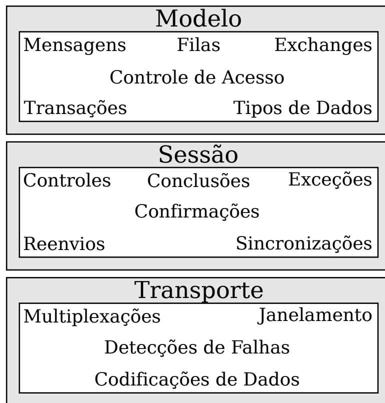
Formatos suportados de seriação:

- Java, SBinary, JSON e Protobuf

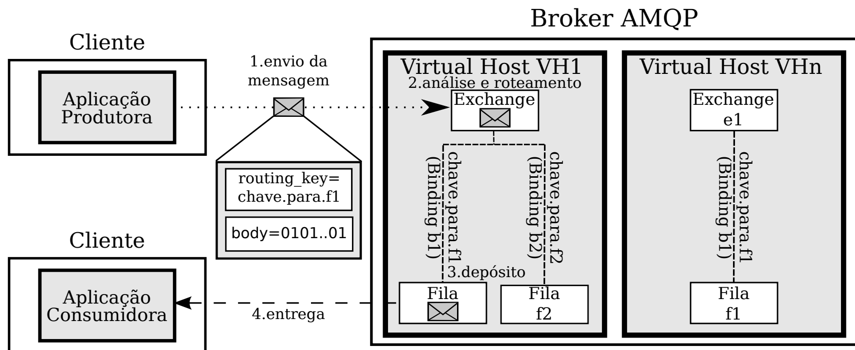
Atores remotos



Advanced Message Queuing Protocol



Advanced Message Queuing Protocol



Advanced Message Queuing Protocol

Algumas implementações:

- Apache Qpid
- ZeroMQ
- RabbitMQ

Utilizamos neste trabalho a implementação de código aberto RabbitMQ

Pontes AMQP

Uma ponte AMQP é:

- Um componente intermediário entre o *message broker* e a camada de suporte a atores remotos
- O componente responsável pela criação de objetos como filas e *exchanges* no *message broker*
- O componente responsável pelo envio e recebimento de mensagens

Pontes AMQP

Principais características:

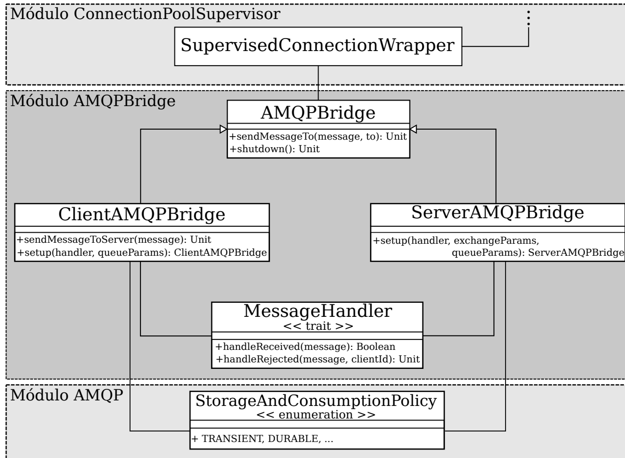
- Rotulação de entidades por nome (e.g.: node1)
- Suporte para acesso concorrente nos canais do RabbitMQ via hierarquia de atores
- Suporte para separação do fluxo de leitura/escrita das mensagens em canais distintos
- Suporte para criação de objetos duráveis, transientes ou exclusivos com auto remoção

Pontes AMQP

Principais características (cont):

- Dois tipos de pontes AMQP para acesso ao *message broker*:
    1. ServerAMQPBridge: Define o comportamento das entidades com papel de servidor
    2. ClientAMQPBridge: Define o comportamento das entidades com papel de cliente
- Recebimento de mensagens via MessageHandler associado às pontes

Pontes AMQP

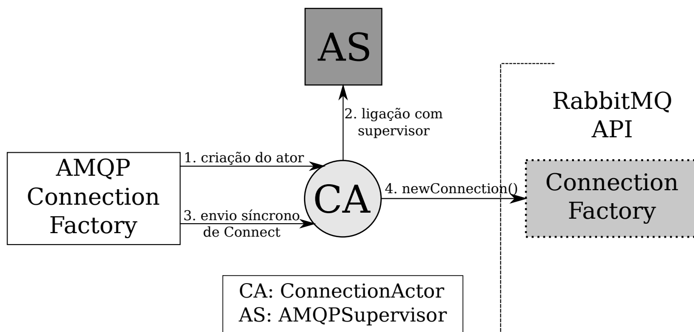


Pontes AMQP – SupervisedConnectionWrapper

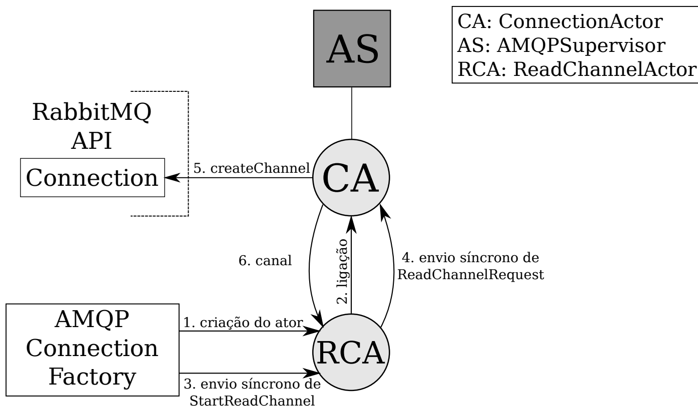
Métodos da classe SupervisedConnectionWrapper:

```
class SupervisedConnectionWrapper(
      connection: ActorRef,
      readChannel: ActorRef,
      writeChannel: ActorRef) {

  def clientSetup(setupInfo: RemoteClientSetup) { ... }
  def serverSetup(setupInfo: RemoteServerSetup) { ... }
  def publishTo(exchange: String, routingKey: String, message: Array[Byte]) { ... }
  def close() { ... }
}
```

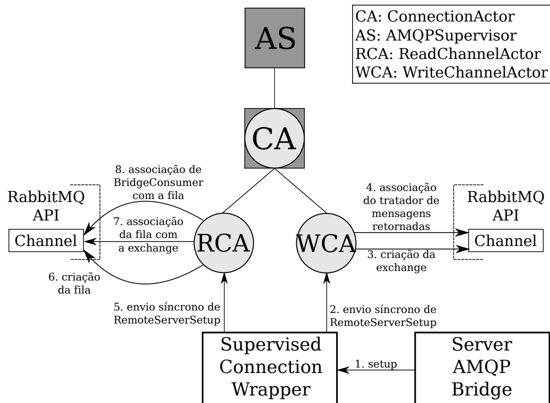
Pontes AMQP – SupervisedConnectionWrapper



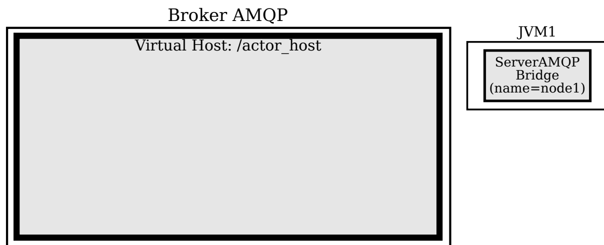
Pontes AMQP – SupervisedConnectionWrapper



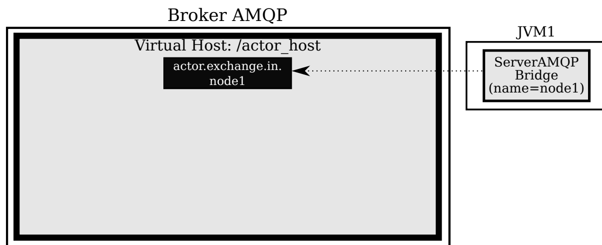
Pontes AMQP – Estrutura



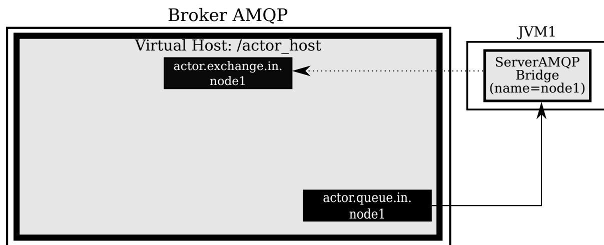
Pontes AMQP – Estrutura



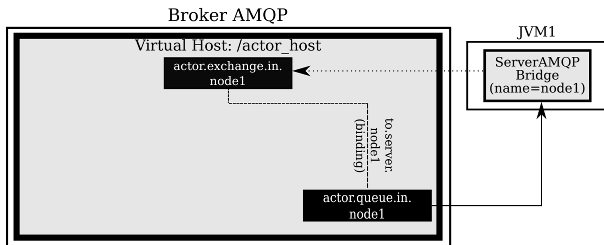
Pontes AMQP – Estrutura



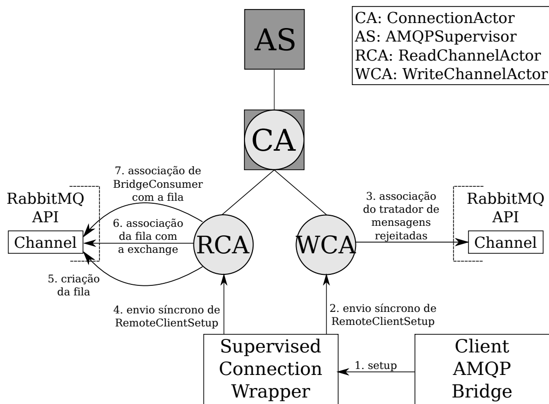
Pontes AMQP – Estrutura



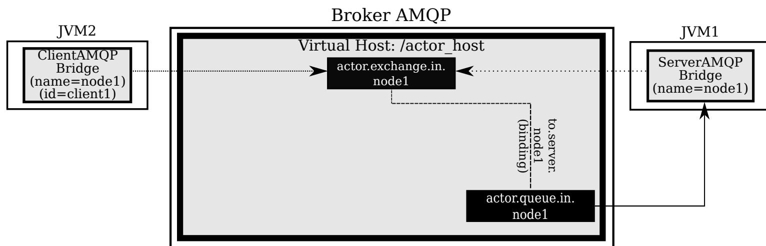
Pontes AMQP – Estrutura



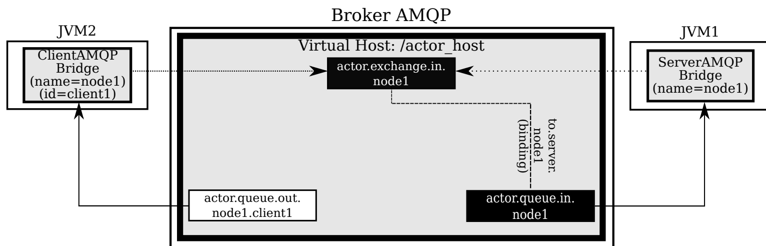
Pontes AMQP – Estrutura



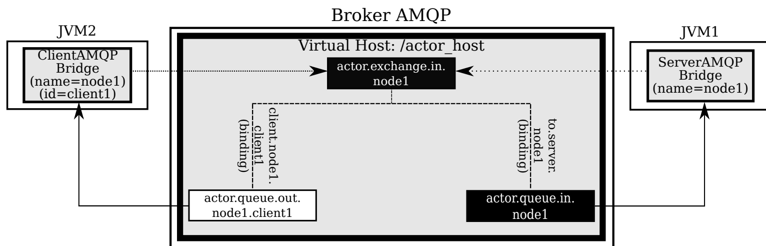
Pontes AMQP – Estrutura



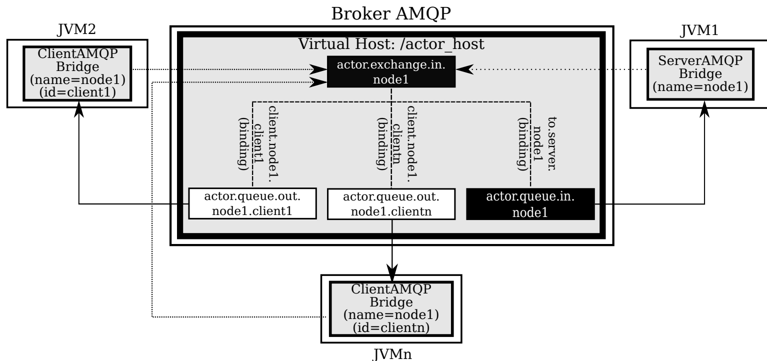
Pontes AMQP – Estrutura



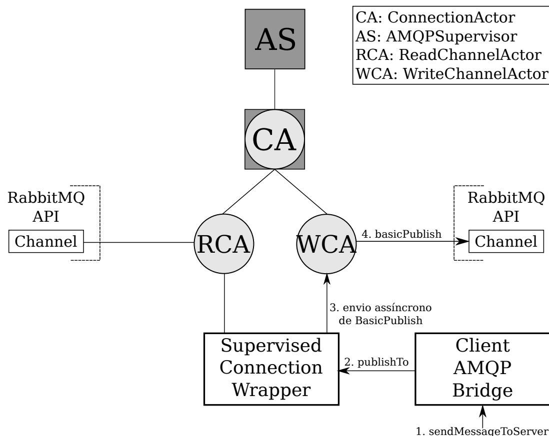
Pontes AMQP – Estrutura



Pontes AMQP – Estrutura



Pontes AMQP – Envio

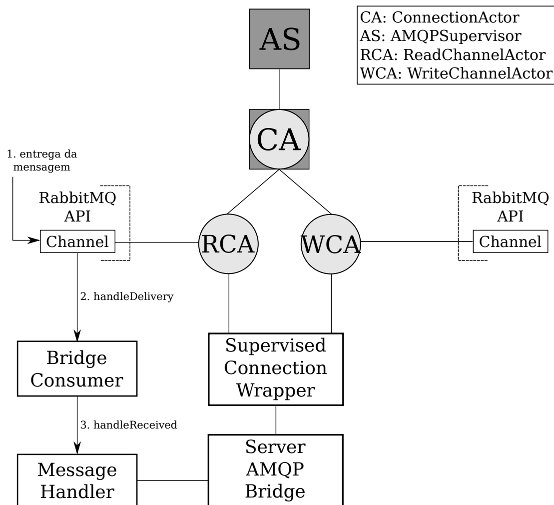


Pontes AMQP – Envio

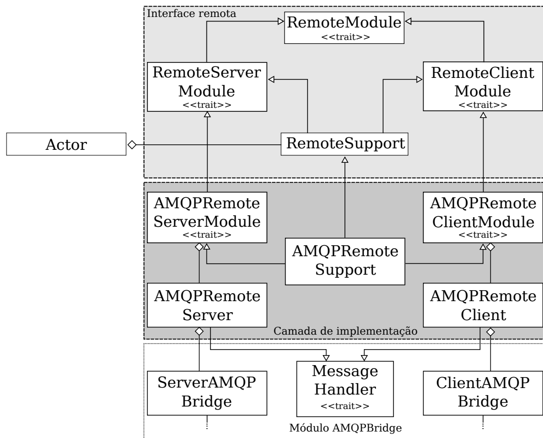
Envio via WriteChannelActor:

```
...
case BasicPublish(exchange, routingKey, mandatory, immediate, message) =>
{
  channel.foreach {
    ch => ch.basicPublish(exchange, routingKey, mandatory, immediate, null, message)
  }
}
...
```

Pontes AMQP – Recebimento



Novos componentes no Akka



Adaptações e alterações necessárias

1. Adaptação dos valores host e porta recebidos nos métodos da biblioteca do Akka para o padrão @host:porta

Exemplo:

Actor.remote.start("localhost", 2552) -> name: localhost@2552

Adaptações e alterações necessárias

1. Adaptação dos valores host e porta recebidos nos métodos da biblioteca do Akka para o padrão @host:porta
2. Novas entradas na seção remote do arquivo de configurações do Akka:
   - Informações para conexão com o *message broker*
   - Políticas de armazenamento e compartilhamento de canais
   - Identificador utilizado na criação das pontes cliente

Adaptações e alterações necessárias

```
remote {
  ...
  layer = "akka.remote.amqp.AMQPRemoteSupport"
  amqp {
    broker {
      host = "192.168.0.121"
      port = 5673
      virtualhost = "/actor_host"
      username = "actor_admin"
      password = "actor_admin"
    }
    policy {
      storage {
        mode = "PERSISTENT"
        client_id {
          suffix = "MYPLACE"
        }
      }
      connection {
        server = "ONE_CONN_PER_NODE"
        client = "ONE_CONN_PER_NODE"
      }
    }
  }
  ...
}
```

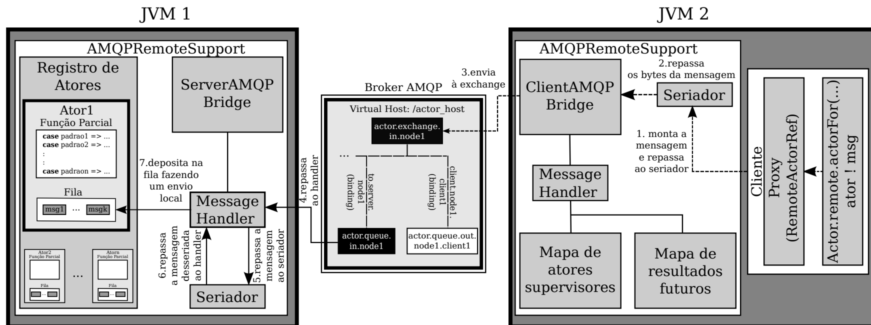
Adaptações e alterações necessárias

1. Adaptação dos valores host e porta recebidos nos métodos da biblioteca do Akka para o padrão @host:porta
2. Novas entradas na seção remote do arquivo de configurações do Akka:
    - Informações para conexão com o *message broker*
    - Políticas de armazenamento e compartilhamento de canais
    - Identificador utilizado na criação de pontes cliente
3. Novo campo no formato das mensagens para que o servidor remoto possa identificar o cliente remetente

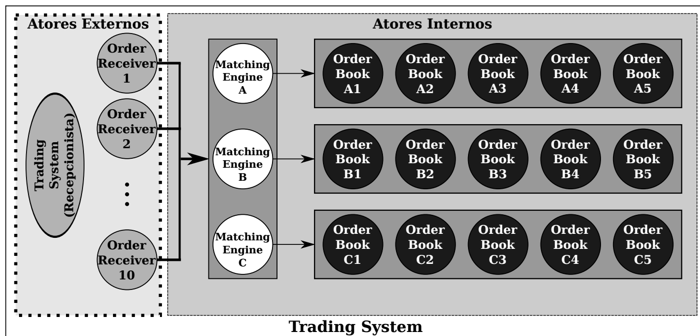
Adaptações e alterações necessárias

```
message RemoteMessageProtocol {
  required UuidProtocol uuid = 1;
  required ActorInfoProtocol actorInfo = 2;
  required bool oneWay = 3;
  optional MessageProtocol message = 4;
  optional ExceptionProtocol exception = 5;
  optional UuidProtocol supervisorUuid = 6;
  optional RemoteActorRefProtocol sender = 7;
  repeated MetadataEntryProtocol metadata = 8;
  optional string cookie = 9;
  optional string remoteClientId = 10; // <- identificador do cliente
}
```

Fluxo de envio de mensagens



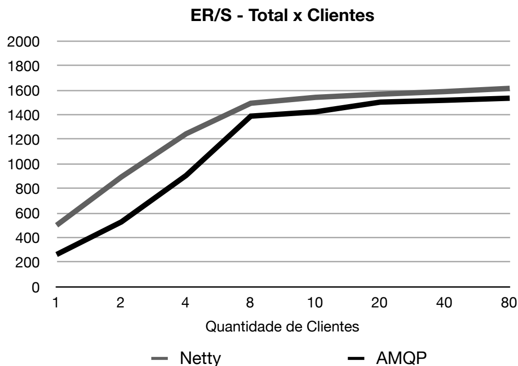
Trading System



Rede de baixa latência

Impl.	Clientes	ER/S	25% (μs)	50% (μs)	75% (μs)	95% (μs)	Dur. (s)
Netty	1	501	1364	1433	1968	3266	59.85
AMQP	1	263	3283	3361	3537	4550	113.91
	Variação	**-47.50%**	**140.69%**	**134.54%**	**79.73%**	**39.31%**	**54.06 (90.33%)**
Netty	2	894	1770	1796	1836	3290	33.57
AMQP	2	528	3234	3359	3552	4208	56.83
	Variação	-40.99%	82.71%	87.03%	93.46%	27.90%	23.26 (69.33%)
Netty	4	1245	2070	2517	2726	4214	24.09
AMQP	4	906	3248	3614	4134	5088	33.12
	Variação	-27.31%	56.91%	43.58%	51.65%	20.74%	9.03 (37.48%)
Netty	8	1494	3274	3771	4745	6168	20.07
AMQP	8	1389	3567	4149	5168	6425	21.60
	Variação	-7.10%	8.95%	10.02%	8.91%	4.17%	1.52 (7.60%)
Netty	10	1542	4005	4330	5704	7041	19.45
AMQP	10	1424	4396	5107	6136	7610	21.06
	Variação	-7.72%	9.76%	17.94%	7.57%	8.08%	1.61 (8.31%)
Netty	20	1568	8206	9646	10146	12015	19.13
AMQP	20	1503	8728	9964	10888	13430	19.96
	Variação	-4.15%	6.36%	3.30%	7.31%	11.78%	0.82 (4.33%)
Netty	40	1589	17616	18183	19436	21350	18.82
AMQP	40	1518	18496	19525	20842	23360	19.75
	Variação	-4.41%	5.00%	7.38%	7.23%	9.41%	0.87 (4.64%)
Netty	80	1615	35403	36541	37863	40041	18.57
AMQP	80	1535	37915	39399	40943	43496	19.54
	Variação	**-5.02%**	**7.10%**	**7.82%**	**8.13%**	**8.63%**	**0.97 (5.24%)**

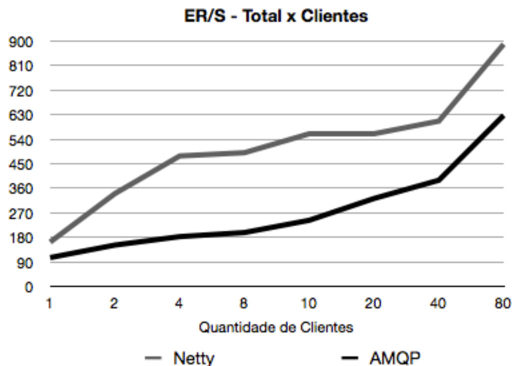
Rede de baixa latência



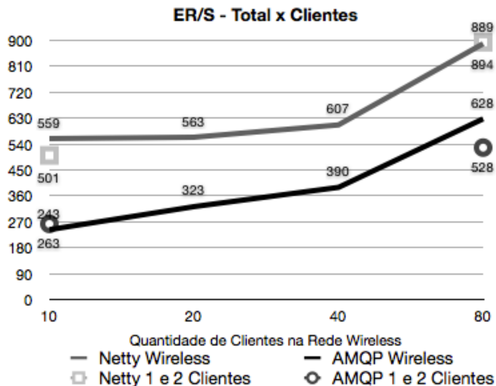
Rede de alta latência

Impl.	Clientes	ER/S	25% (μs)	50% (μs)	75% (μs)	95% (μs)	Dur. (s)
Netty	1	163	3387	4279	6265	13403	183.87
AMQP	1	106	5998	7223	9624	17360	283.92
	Variação	-34.97%	77.09%	68.80%	53.62%	29.52%	100.05 (54.41%)
Netty	2	341	3418	4113	5635	11481	87.855
AMQP	2	152	8963	10784	14077	22309	196.815
	Variação	**-55.43%**	**162.23%**	**162.19%**	**149.81%**	**94.31%**	**108.96 (124.02%)**
Netty	4	479	5248	6334	8219	13892	62.5275
AMQP	4	183	16231	18823	22940	34045	164.04
	Variação	-61.80%	209.28%	197.17%	179.11%	145.07%	101.51 (162.35%)
Netty	8	491	10528	12692	15780	23123	61.095
AMQP	8	198	26918	33224	41392	58513	151.73625
	Variação	-59.67%	155.68%	161.77%	162.31%	153.05%	90.64 (148.36%)
Netty	10	559	12737	14918	18099	25705	53.649
AMQP	10	243	29653	34970	41936	56755	122.673
	Variação	**-56.27%**	**132.81%**	**134.41%**	**131.70%**	**120.79%**	**69.02 (128.66%)**
Netty	20	563	22392	26879	32957	47837	53.23
AMQP	20	323	47710	55291	64509	84108	92.92
	Variação	-42.63%	113.07%	105.70%	95.74%	75.82%	39.69 (74.58%)
Netty	40	607	41155	49748	60805	83042	49.33
AMQP	40	390	62679	77785	98463	150803	74.21
	Variação	-35.75%	52.30%	56.36%	61.93%	81.60%	24.87 (50.42%)
Netty	80	889	36654	39031	50225	103695	33.41
AMQP	80	628	66128	77563	94990	139215	42.34
	Variação	**-29.36%**	**80.41%**	**98.72%**	**89.13%**	**34.25%**	**8.93 (26.74%)**

Rede de alta latência (wireless)



Fluxo nas redes de alta e baixa latência



Trabalhos futuros

- Melhoria no tratamento de erros nas pontes AMQP
  - Aproveitar a hierarquia de supervisão já definida nas pontes AMQP para o tratamento de erros
  - Implementar suporte a reconexão e restauração do estado dos atores relacionados a ponte AMQP no caso de desconexões ou falhas de rede
- Experimentos em um ambiente de computação em nuvem
  - Fazer as alterações necessárias para que nossa implementação de atores remotos possa ser implantada em uma infraestrutura de computação em nuvem
  - Fazer uma avaliação experiemental do *Trading System* na nuvem utilizando uma quantidade de clientes bem maiores do que as utilizadas em nossos experimentos

Conclusões

Conclusão:

As duas classes de sistemas estudadas neste trabalho possuem boa sinergia.

Conclusões

O emprego de um *message broker* ainda traz outros benefícios, como por exemplo:

- Mensagens duráveis: envios de mensagens para entidades que estão temporariamente fora de execução
- Ferramentas administrativas: a maioria dos *message brokers* possuem ferramentas para administraçã e monitoramento
- Simplificação de acesso: com a identificação por notesm, apenas o endereço IP do *message broker* precisa ser conhecido

Conclusões

- Ainda que o modelo de atores venha ganhando muita atenção em programas de alta concorrência, o modelo não tem emprego difundido em ambientes corporativos
- Nossa expectativa é que a integração com sistemas orientados a troca de mensagens contribua para a adoção do modelo de atores em tais ambientes

Fim

Obrigado!