Names

March 28, 2020

Outline

- The Problem
- Naming Concepts
- Name Spaces
- Name Resolution
- Addional Reading

Server/Object Location

Problem: How does a client know where is the server?

Solution: Not one, but several alternatives:

- *hard coded*, seldom;
- via program arguments: more flexible, but ...;
- via configuration file;
- via *broadcast*/*multicast*;
- via a location/name service:
  - local, e.g. *rmiregistry*.
  - *global*.

Addresses vs. Names

- Names are ... sequences of symbols (bits/characters/...) that refer to entities/objects.
- In the labs, we have used IP addresses (and ports)
- Addresses are **names** of **access points**. Or as Shoch put it:

  *The **name** of a resource indicates **what** we seek,*
  *an **address** indicates **where** it is,*
  *(and a **route** tells us **how** to get there.)*

- Addresses have some limitations:
  - Addresses often are location dependent and change frequently
    - E.g. when a service is moved from one computer to another
- Names have some advantages over addresses:
  - They can be human-friendly.
  - They can hide both complexity and dynamics
    - E.g. they can hide access point changes
- Naming is a layer of indirection
  - Ultimately you need an address to access/operate on an object

Identifiers

- An **identifier** is a name with 3 properties:
  1. an identifier refers to one entity at most;
  2. an entity has at most one identifier;
  3. an identifier refers always to the same entity (it is never reused).
- Identifiers provide a mean to refer to an entity in a precise way, independently of its access points.
- Examples?
  - From the "real" world?
  - From the "virtual" world?

Pure Names

- Are names that contain no information whatsoever about what they refer to:
  - Not only about location, but about anything else
  - They do not commit the system to anything
  - They are useful only for comparison

Problems/challenges of pure names

- where to look them up to find out information about them?
- how do you know that an object does not exist? How can a global search be avoided?
- how to engineer uniqueness reliably in a distributed system?

Problems/challenges of impure names

- what if the information yielded by the name, e.g. location, is not valid anymore?
  - This is specially relevant for mobile systems, and requires appropriate solutions

Bindings, Contexts and Name Resolution

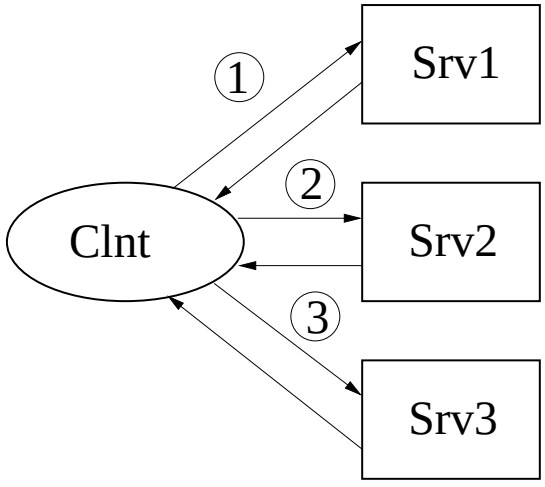
- A **binding** is a mapping from a name to an object/entity (usually identified by a lower-level name, e.g. address)
- A context/name space is a set of **bindings**
- A name space defines:
  - the syntax and structure (flat vs. hierarchical) of a name
  - the rules to find a binding of a name (**name resolution**)
- **Name resolution** is the process of finding a binding for a name
- A name is always resolved in the context of its name space:

file name	->	OS filesystem
Java program variable	->	JVM executing the program
ISBN of a publication	->	ISBN (Intern.Standard Book Number)
Car license plates	->	national/regional license plate regist

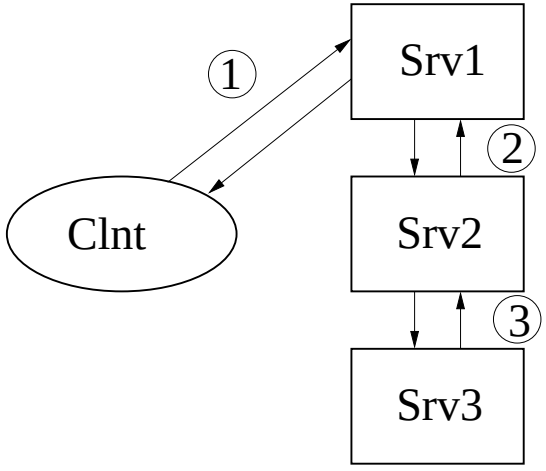
Name Resolution in a Distributed System

- Usually, name resolution is done with the help of a name service
- In small scale distributed systems, name resolution requires only one server:
  - E.g., the `rmiregistry`
- In distributed systems of larger scale, name resolution may require more than one server. In this case, name resolution can use one of 3 strategies:
  1. Iterative
  2. Recursive.
  3. Transitive.

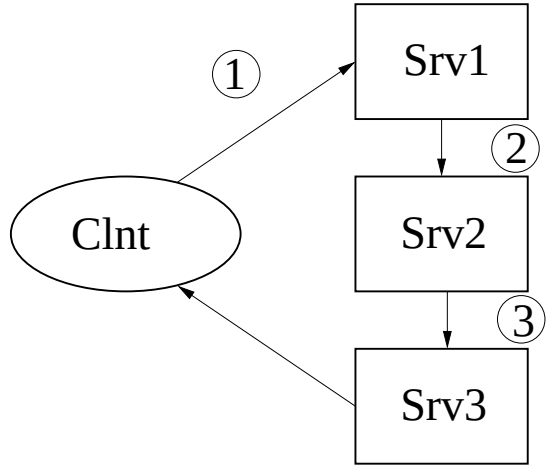
Name Resolution: Strategies



Iterative



Recursive



Transitive

- Recursive name resolution:
  - Allows for caching at servers
    - This may make resolution more efficient (with lower communication costs)
  - But, it:
    - requires servers to keep state
    - makes it harder to set the values of timeouts
- Transitive name resolution also makes it harder for the client to set a timeout value

Name Resolution and *Closure Mechanism*

Names are resolved always in a context

Problem

- How do you get a context that you can use to resolve a name?
  - How do you get a "remote reference to the `rmiregistry`"?
  - How to start the name resolution of a name of a file system: i.e. where is the root directory?
  - How to find the IP address of a DNS server to resolve a DNS name?

Response

Use a **closure mechanism**

- Typically this is an *ad-hoc* and simple solution.

Hierarchical Name Spaces

- Most name spaces have a hierarchical structure:
  - OS filesystem
  - Domain Name System (DNS)
  - Postal addresses
  - Car license plates are resolved in another context – per country, region etc.
- A hierarchical structure simplifies:
  - the assignment;
  - the resolution

  of names
- Allows to partition a name space into naming domains
  - Often, a naming domain has an administrative authority for assigning names within it
  - An administrative authority may delegate name assignments for sub-domains (e.g. in DNS)

Additional Reading

- Chapter 5 of van Steen and Tanenbaum, *Distributed Systems, 3rd Ed.*
  - Section 5.1: *Names, Identifiers and Addresses*
  - Section 5.3: *Structured Naming*
- J. Saltzer, *On the Naming and Binding of Network Destinations*, in RFC 1498, 1993

1. Introdução

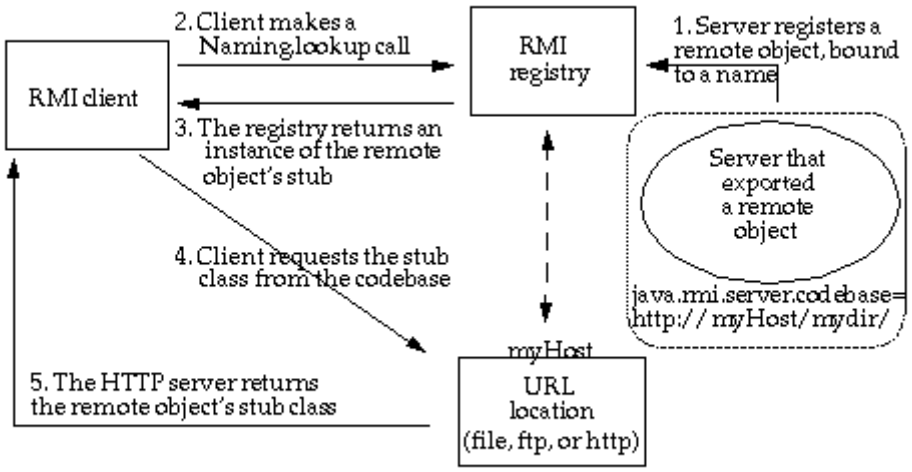
Na aula TP de SDist desta semana, trabalharemos no exercício apresentado em [1]. O principal objetivo desta aula é que vocês entendam os conceitos de base da comunicação sobre Java RMI.

No final de [1] encontram um tutorial útil sobre comunicação Java RMI (este [2]).

Tanto o exercício desta aula como o exemplo do tutorial Oracle [2], concentram-se no desenvolvimento de uma interface RMI entre um cliente e um servidor que poderá depois, de alguma forma, ser reutilizada no primeiro projecto avaliado.

(especificamente na comunicação entre o *TestClient* e o *Peer*, caso pretendam implementar essa comunicação usando o RMI).

2. Comunicação RMI



Na comunicação RMI, a interacção entre um servidor e um cliente é assistida (numa fase inicial) por outra aplicação, i.e. o *rmiregistry*.

O *rmiregistry* é um serviço de registo de nomes (nomes de outros serviços) para *bootstrap* da comunicação entre clientes e servidores. O *rmiregistry* regista assim uma associação entre nomes de serviços e os *remote objects* que os implementam, mais especificamente entre nomes de serviços e as referências para esses *remote objects* (designadas como os *stubs* desses *remote objects*).
Os servidores procedem ao registo dessas associações, i.e. inscrevem no *rmiregistry* pares de *nome-de-serviço/stub*.
Os clientes, tanto no *host* local como em *hosts* remotos, contactam o *rmiregistry* para obterem os *stubs* de serviços específicos, que depois empregam para invocar tais serviços.

No entanto, os *stubs* (ou as referências para *remote objects*) registados no *rmiregistry* não são suficientes para que o cliente possa aceder ao servidor. O cliente precisa dos ficheiros *.class* com a definição da classe do *stub* (mais especificamente a classe que implementa o serviço remoto) e das restantes classes de que esta necessite.

No contexto de Java RMI, o local/serviço onde estes ficheiros *.class* estão armazenados, para serem disponibilizados ao lado cliente, chama-se *codebase* (a qual é independente do *rmiregistry*)

A *codebase* pode ser assim definida como a fonte a partir da qual os ficheiros *.class*, com as implementações dos artefactos de comunicação, podem ser carregados para uma máquina virtual. Para obterem uma informação mais detalhada sobre a *codebase* e sobre como o lado do servidor deve lidar com ela, podem consultar a página indicada em [3].

A operação básica da comunicação em RMI Java é a seguinte (de [3]):

1. O servidor registra um *remote object*, associando-o a um nome, no *rmiregistry*. O que é registado no *rmiregistry* não é o efectivo *remote object*, é apenas uma referência para um *remote object* (ou *stub*). Esse *stub* é gerado de forma dinâmica (programaticamente) no servidor.
   Para que o *stub* possa ser executado no lado cliente é necessário que este consiga obter a informação em falta no *stub* (os ficheiros *.class* com as definições das classes que implementam o serviço remoto) que está armazenada na *codebase*. O que acontece em Java RMI é que a localização dessa *codebase*, é anotada no *stub* quando este é produzido (é daí que o cliente a obtém).
   Assim aquando de construção dinâmica do *stub* pelo lado servidor, a sua VM tem de saber onde está a *codebase*. Para isso esta propriedade, da VM em questão, deve ser previamente estabelecida (configurando a propriedade *java.rmi.server.codebase* na linha de comando ou programaticamente).
   Só depois de estar configurada a referida propriedade, e gerado o *stub* é que se pode proceder ao registo efectivo.

2. O cliente solicita, ao *rmiregistry*, uma referência para um *remote object* específico.

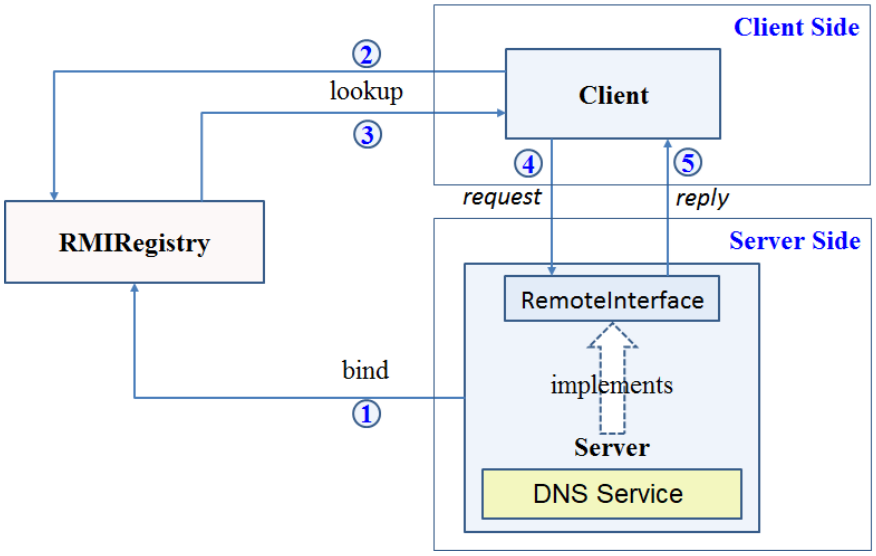
3. O *rmiregistry* retorna a referência (*stub*) em questão.

4. O cliente trata de obter a(s) definição(s) da(s) classe(s) da necessárias à execução do *stub*. Se estas definições (*.class*) puderem ser obtidas da *classpath* do cliente (que é sempre pesquisada antes da *codebase*), o cliente carregará a classe localmente. Caso contrário o cliente tentará obter as definições de classes da *codebase*, empregando o valor da *codebase* (URL) que está anotado no próprio *stub*.

5. A definição de classe para o *stub* (e quaisquer outras classes necessárias) é retornada ao cliente.

6. Neste ponto, o cliente tem todas as informações necessárias para invocar os métodos oferecidos pelo *remote object*. A instância do *stub* funciona assim como um *proxy* para o *remote object* que executa do lado servidor. Desta maneira, e diferentemente de um *applet* que recorre a uma *codebase* para obter código remoto e executa-lo na sua VM local, o cliente Java RMI recorre a uma *codebase* para obter código remoto que desencadeia a execução de código noutra VM potencialmente remota.

3. Exercício do Lab 3



A imagem acima apresenta uma descrição simples da arquitetura que deverão implementar no lab 3 (bem como no tutorial da Oracle [2]). É muito próximo daquilo que já foi apresentado na imagem anterior, e assim, também é a sua explicação.

As etapas para desenvolver uma aplicação cliente-servidor utilizando Java RMI são as seguintes:

1. Definir a interface remota oferecida pelo servidor. Terão assim de definir um interface que estenda o interface *java.rmi.Remote*. Estender este interface implica as seguintes restrições:
   a) todos os métodos declarados (no interface do serviço) devem lançar uma excepção do tipo *java.rmi.RemoteException* (essa excepção é lançada pela JVM no caso de problemas de comunicação entre as JVMs);
   b) os argumentos e valores de retorno de todos os métodos declarados devem implementar a interface *java.io.Serializable*.

2. Desenvolver a classe do servidor, ou seja, a classe que implementa a interface definida no ponto anterior. Esta classe (ou a classe que inicia a execução do servidor) deve efectuar os seguintes passos:
   a) fazer, programaticamente, o set da propriedade *java.rmi.server.codebase* com o valor da *codebase* (para que este valor seja conhecido pela VM e possa ser anotado no *stub* aquando da sua criação);
   b) instanciação do "*remote object*". Para isso o primeiro passo é criar uma instância da classe servidor (usando o seu constructor). Depois é necessário "exportar" o *remote object*, ou seja, colocar o serviço a correr numa determinada porta e produzir o respectivo *stub*. As duas tarefas são realizadas pelo método método *static export(….)* da classe *java.rmi.server.UnicastRemoteObject.* O valor de retorno deste método é o *stub*;
   c) registo do *stub*, associado a um nome, no *rmiregistry*. Para isso existem duas opções:
      a. empregar os métodos *static bind(..)* ou *rebind(…)* da classe *java.rmi.Naming*. Estes permitem efectuar o registro de uma interface remota no *rmiregistry*;
      b. empregar o método *getRegistry()* da classe *java.rmi.registry.LocateRegistry*. Obtém-se assim uma referência a um objecto que implementa o interface *java.rmi.registry.Registry* (ou seja, retorna um objeto que representa o *rmiregistry*), a qual oferece os métodos descritos acima *bind()* e *rebind()*.

Em ambas as situações é preferível usar *rebind(…)*, pois *bind(…)* lançará uma excepção caso nome registrado anteriormente (ou seja, uma *String*) seja reutilizado. *rebind(…)* reassocia o nome ao "novo" *remote object*.

3. Desenvolver o cliente. Este deverá executar os seguintes passos:
   a) obter, do *rmiregistry*, o *stub* de acesso ao serviço remoto, ou seja, fazer o *lookup* do serviço. Para isso há duas alternativas (de forma semelhante ao que é descrito em 2c):
      a. empregar o método *static lookup(..)* da classe *java.rmi.Naming*. Este permite a obtenção de um *stub* para um *remote object* registrado no *rmiregistry*.
         Neste caso o argumento do método *lookup(…)* deve assumir o seguinte formato: //<host_name> [: < port_number >] / <obj_name> - em que < port_number > é o número da porta usada pelo *rmiregistry* (por defeito é 1099);
      b. empregar o método *getRegistry()* da classe *java.rmi.registry.LocateRegistry*. Obter assim uma referência a um objecto que implementa o interface *java.rmi.registry.Registry* e assim ter acesso ao método *lookup(…)*.
         Neste caso o argumento *String* do método *lookup(…)* pode ser apenas <obj_name>, pois os valores de <host_name> e de < port_number > podem ser especificados usando os argumentos transmitidos ao método *getRegistry()*;

      Ambos os métodos retornarão uma instância de um objecto que implementa o interface do serviço definido no ponto 1.
   b) após os passos anteriores o cliente estará então pronto para invocar os métodos remotos oferecidos pelo serviço de DNS de acordo com a assinatura dos mesmos. Essa invocação pode levar a uma excepção do tipo *java.rmi.RemoteException*, e deve, portanto, ser feita dentro de um bloco *try-catch*.

Notas relativas à execução do lab 3 (execução de rmiregistry, servidor e cliente):

1. O *rmiregistry* deve estar a correr antes de se executar o servidor. Portanto, primeiro corre-se o *rmiregistry*, depois o servidor e depois o cliente;

2. O *rmiregistry* pode ser executado a partir da linha de comando ou programaticamente. Recomenda-se a primeira alternativa;

3. Para poder executar o *rmiregistry* (a partir da linha de comando), o seu executável deve estar na *classpath* da *shell*. É necessário verificar se o diretório *bin* da instalação Java (onde se encontra o *rmiregistry.exe*) foi adicionado à variável de ambiente *Path* (esse é o nome da variável no Windows);

4. Para simplificar a execução do *rmiregistry* este deve ser executado no mesmo diretório em que o servidor é executado (assumindo que este é executado na base da árvore directorial que contém os *.class*).
   Executar o *rmiregistry* no diretório em questão é vantajoso pela seguinte razão: se nada for especificado, o *rmiregistry* assume que seu diretório de execução é também a *codebase*. Dessa maneira, se o *rmiregistry* for executado no mesmo diretório do servidor, o primeiro encontrará automaticamente os ficheiros *.class* com a implementação do serviço;

5. O *rmiregistry* também pode ser executado a partir de outro diretório. Nesse caso, a propriedade *java.rmi.server.codebase* deve ser configurada. Para isso, no momento da inicialização do *rmiregistry*, a codebase deve ser especificado (por exemplo, *rmiregistry -J-Djava.rmi.server.codebase=file:///a/b/SDist_2018/L03/bin/* onde file:///a/b/SDist_2018/L03/bin/ é um valor a adaptar às características da vossa implementação;

6. No que respeita à execução do servidor, caso a propriedade *java.rmi.server.codebase* não seja configurada programaticamente, isso deverá então ser feito na linha de comandos tal como é explicado no ponto 5:

7. Em relação ao cliente, poderá ser necessário configurar a política de segurança para que a VM carregue "executáveis" remotos. Para isso recomenda-se a consulta do ponto D da seção 6.0 da referência [3].

4. Referências

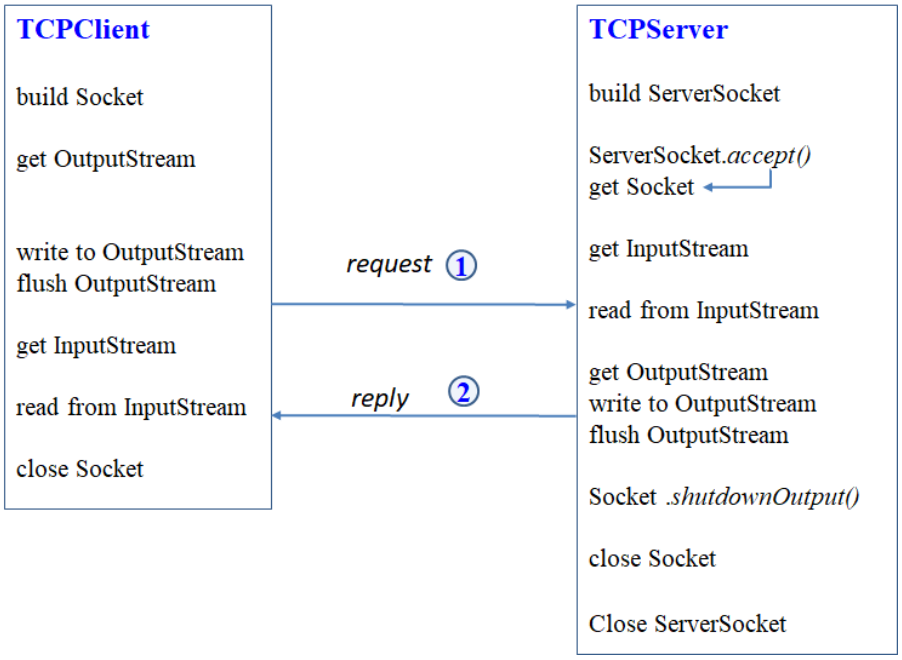
[1] Lab on RMI – https://web.fe.up.pt/~pfs/aulas/sd2020/labs/l03/rmi_l03.html

[2] Getting Started Using Java RMI, Oracle Tutorial, https://docs.oracle.com/javase/6/docs/technotes/guides/rmi/hello/hello-world.html

[3] Dynamic code downloading using Java RMI, Oracle, https://docs.oracle.com/javase/6/docs/technotes/guides/rmi/codebase.html

Exercício sobre TCP

1. Arquitectura Geral



2. Objectivos para a Aula 6

O objetivo da 6ª aula TP é permitir a vossa aquisição de conhecimentos no contexto da comunicação TCP (empregando a API Java para esse efeito).

O guião do exercício a ser implementado na aula está disponível em https://web.fe.up.pt/~pfs/aulas/sd2020/labs/l04/tcp_l04.html
(trata-se de mais uma versão do lab1, desta vez empregando TCP na comunicação entre cliente e servido)

No contexto do *assigment 1* a API Java para TCP poderá ser empregue:

- na comunicação entre o *TestClient* e os *Peers*, caso não consigam/queiram desenvolver esta interface em RMI;
- no *enhancement* to "*Chunk Restore Protocol*" (secção 3.3 do guião do *assigment 1*);

3. Notas Sobre a API Java para TCP

As notas abaixo estão de acordo com as transparências apresentadas pelo Professor Pedro Souto na aula teórica, e presentes primeiro "link" no final do guião.

1) A API de Java distingue entre:

- sockets usados para aceitar conexões – deve usar-se uma instância da classe *java.net.ServerSocket* para ficar permanentemente á escuta de ligações TCP numa determinada porta;
- sockets usados para transferir dados – são usadas instâncias da classe *java.net.Socket* para a efectiva leitura e envio de informação para o canal TCP

2) Embora a classe *Socket* seja usada para a troca de informação, esta não oferece métodos para o fazer diretamente. Fornece antes os métodos *getInputStream()* e *getOutputStream()*. Estes retornam referências para objetos (que implementam as classes abstratas *InputStream* e *OutputStream*), os quais permitem a efectiva troca de informação.

3) Os objectos retornados pelos métodos acima mencionados são bastante básicos e permitem apenas transferir *arrays* de bytes.
Pode-se, no entanto, usar estes *streams* básicos para construir *streams* com mais funcionalidades (classes da *package java.io.*). Podem assim criar-se *streams* capazes de efectuar a escrita/leitura de:
- texto formatado (*PrintWriter*/*BufferedReader*),
- tipos primitivos de Java (*DataOutputStream*/*DataInputStream*)
- objetos Java serializáveis (*ObjectOutputStream*/*ObjectInputStream*)

Notem que é importante que os *streams* (instâncias de classes da package *java.io.*) empregues para enviar mensagens numa das extremidades, e recebê-las na outra, sejam os correspondentes (os pares acima indicados). Assim, é pouco útil, por exemplo, usar um *ObjectOutputStream* para enviar dados, se na outra extremidade se usa um *BufferedReader*.

No exercício da aula desta semana fará sentido usarem o par *PrintWriter*/*BufferedReader*. O emprego destas classes permite estruturar as mensagens como linhas e troca-las como tal, usando os métodos *println()* e *readLine()* daquelas classes.

No caso *assignment*, o emprego de TCP será para a transferência de *chunks* (secção 3.3) e assim o uso das classes *PrintWriter* e *BufferedReader* não é adequado. Deverá antes ser suficiente o uso dos *streams* básicos, i.e. *OutputStream* e *InputStream*.

4) Algumas das classes que implementam *streams*, (e.g. *PrintWriter*), fazem *buffer* dos dados antes de os enviarem. Assim podem não enviar os dados que são "escritos" neles imediatamente para o socket, o que poderá levar a comportamentos inesperados. É por isto conveniente que se faça *flush* do *stream* onde se está a escrever (invocando o método *flush()*).
No caso da classe *PrintWriter* existe um construtor que permite configurá-lo de forma a que este faça *flush* automaticamente sempre que nele é escrita uma linha.

5) Para garantir que ao fechar uma conexão (método *close()* das classes *Socket*) não se perdem dados (devido, por exemplo, à questão abordada no ponto anterior), antes desse fecho efectivo deverá chamar-se o método *shutdownOutput()*, e só depois então o metido *close()*.

Projecto 1 – Notas Sobre Concorrência

1. Introdução

Este documento sugere uma possível via para uma implementação altamente concorrente e escalável do Projeto 1. Usa-se como exemplo a implementação do *Chunk Backup Protocol* (mensagem **PUTCHUNK**).

O que vos propomos é que desenvolvam uma implementação gradual do protocolo e, através de iterações sucessivas, melhorem a simultaneidade/concorrência e a escalabilidade da vossa implementação. O que é apresentado nas secções seguintes segue precisamente este método de desenvolvimento gradual.

Este documento não cobre, obviamente, todos as soluções possíveis.
Se se sentirem confiantes no vosso trabalho de implementação com *threads* podem saltar diretamente para a seção 3.

2. Implementação *Single Threaded*

Suposição: nesta implementação considera-se que é apenas necessário executar uma instância do protocolo de cada vez.

No final da implementação desta versão (simplificada) do protocolo, serão capazes de efectuar o *backup* de ficheiros com um único *chunk*, ou com vários deles, desde que tratem do *backup* de um *chunk* apenas depois de terminar o processo de *backup* do *chunk* anterior.

A ideia de base consiste em usar um único *thread* para cada um dos seguintes:
- *peer* iniciador
- *peer* não iniciadores

Simplificação: dado que esta implementação se trata apenas de uma versão preliminar, não há necessidade de medir com precisão o intervalo entre as retransmissões (da mensagem **PUTCHUNK**). Em vez de se medir o intervalo entre retransmissões, mede-se antes a duração do silêncio no meio (canais), ou seja, desde a transmissão do **PUTCHUNK** mais recente ou desde a recepção do **STORED** mais recente.

Nesta versão simplificada do protocolo pode fazer-se uso de *DatagramSocket.setSoTimeout()* e *Thread.sleep()* para medir a passagem do tempo (e implementar tempos de espera), e usar um único *thread*, tanto para o *peer* iniciador como para os outros *peers*.

3. Um *Thread* por Canal *Multicast*, um Protocolo de cada Vez

Para esta versão da implementação, continuamos a assumir que não há execuções simultâneas de diferentes instâncias de protocolo (*Chunk Backup Protocol*), ou seja, a qualquer momento, há no máximo uma instância de protocolo em execução.

A ideia de base da solução presente consiste em empregar um *thread* por canal *multicast* para tratar da recepção das mensagens enviadas para esse canal. Teremos assim o *Control Receiver thread* (que está á

escuta no canal MC) e o *Data Backup Receiver thread* (que está á escuta no MDB). Para o *Chunk Backup Protocol* não é necessário interagir com o canal MDR.

Acrescidamente, cada um dos *threads* mencionados processa as mensagens recebidas, através do respectivo canal, de forma sequencial. Cada *thread* conclui o processamento de uma mensagem antes de iniciar o processamento da mensagem seguinte. Assim, cada um destes *threads* deve executar um *loop* infinito e em cada iteração deste trata de:

- receber uma mensagem no respectivo canal *multicast*;
- processar a mensagem recebida.

O envio de mensagens **PUTCHUNK** pelo *initiator peer* deve ser realizado por um *thread* diferente dos anteriores, i.e. o *thread multicaster*. Desta forma, o *thread multicaster* pode executar *Thread.sleep()* entre o *multicast* de mensagens **PUTCHUNK** consecutivas, implementando o intervalo de espera especificado no guião. Este thread pode ser criado em resposta a uma solicitação do *testClient*, para executar uma instância do *Chunk Backup Protocol*, e pode terminar quando a instância do protocolo acabar de se processar.

Se pretendem implementar a interface entre o *testClient* e o *peer* usando RMI, o *thread multicaster* pode ser o próprio *thread* criado pelo RMI *run-time* para lidar com a operação de *backup* invocada pelo *testClient*.

Caso contrário, ou seja, se pretendem empregar UDP ou TCP para essa interface, será necessário um *thread* adicional para receber os pedidos do *testClient*. Tal como os *Receiver threads*, este *thread* também pode processar os pedidos do *testClient* em série: ou seja, deve concluir o processamento de um pedido antes de iniciar o processamento do próximo (na verdade, se suportasse a execução simultânea de pedidos do *testClient*, violaríamos a suposição, acima expressa, de que existe no máximo uma instância do protocolo em execução a qualquer momento).

Notem que, no *initiator peer*, o *thread multicaster* e o *Control Receiver thread* podem precisar de partilhar informações (i.e. o número de mensagens **STORED** recebidas), para que o primeiro saiba como proceder. Compete-vos assim garantir que na comunicação entre os dois (por exemplo, no acesso aos objectos partilhados para a implementação dessa comunicação) não haja *race conditions* (problemas de concorrência).

Nota: nesta versão da implementação, os *peers* não iniciadores não precisam de processar as mensagens **STORED**. De qualquer das formas, a implementação de uma tal funcionalidade nesses *peers*, não seria assim tão complexa até porque já é algo que deverá ser feito ao nível do *peer* iniciador.

4. Um Thread por Canal M., Múltiplas Instâncias do Protocolo Simultâneas

Nesta versão da implementação considera-se que já deverá ocorrer mais do que uma execução simultânea do protocolo.

Utilizamos a mesma "arquitectura" de *threads* da versão anterior: existe assim um *Receiver Thread* por cada canal *multicast*, e cada um deles conclui o processamento de uma mensagem, antes de iniciar o processamento da próxima.

Embora possa haver mais do que um *thread multicaster* em execução, tanto no colectivo de *peers* como em cada peer individual (ou seja, pode haver vários peers a iniciar *backups* no sistema), cada um desses *threads* participa apenas na execução de uma instância do *Chunk Backup Protocol*.

De forma oposta, os *Receiver threads* podem precisar de processar mensagens relativas a uma instância do protocolo entre o processamento de mensagens relativas a outra instância do protocolo (para poderem lidar com vários pedidos simultâneos do colectivo de *peers*). Os *Receiver threads* devem, assim, empregar um objecto (estrutura de dados para armazenamento e partilha de informação), por instância de protocolo, para manter as informações relevantes para o processamento de mensagens pertencentes a essa instância de protocolo.
Esse objecto deverá ser construído aquando da transmissão/recepção da primeira mensagem da sua instância de protocolo, dependendo se o *peer* é o iniciador (transmissão) ou não (recepção) dessa instância do protocolo. Além disso, o objecto em questão deve ser mantido numa estrutura de dados, partilhada, de onde será recuperado sempre que isso seja necessário para processar cada uma das mensagens subsequentes da sua instância do protocolo de *Backup*. Desta forma, a primeira acção de um *Receiver thread* após receber uma mensagem (do seu canal *multicast*) deverá ser a recuperação (*retrieval*) do objecto (de armazenamento de dados) da instância do protocolo (*Backup*) à qual a mensagem pertence. Após isso poderá então usar as informações armazenadas nesse objeto para processar a mensagem.

Dica: empreguem uma instância da classe *java.util.concurrent.ConcurrentHashMap* para manter os objetos (armazenadores de informação a ser trocada entre *threads*) das diferentes instâncias de cada protocolo (i.e. um *ConcurrentHashMap* por protocolo).

5. Processando Diferentes Mensagens Recebidas Simultaneamente no mesmo Canal

Na versão anterior, as mensagens recebidas através de um determinado canal *multicast* eram tratadas em série.

Para suportar o processamento simultâneo de diferentes mensagens recebidas no mesmo canal, é necessário empregar mais do que um *thread* por canal. A solução típica passa pelo emprego de *Worker threads*: ou seja, empregar um *thread* diferente do *Receiver thread*, para processar as mensagens recebidas. Especificamente, o *Receiver thread* deve lançar um novo *thread* (*Worker* thread) para o processamento de cada mensagem recebida.

5.1. Uma Solução mais Escalável

O problema com a solução anterior é que a criação e o término de *threads* implica algum *overhead*, no qual se incorre uma vez por mensagem. Uma solução mais escalável é usar uma *pool* de *threads*, o que permite evitar a criação de um novo *thread* para processar cada mensagem.

Assim, esta iteração da implementação deverá empregar *thread pools*. Para isso, convém usar a classe *java.util.concurrent.ThreadPoolExecutor*.

6. Não recorrer a *Thread.sleep()*

O recurso a *Thread.sleep()* para implementação de *timeouts* pode levar à existência simultânea de um grande número de *threads* (em espera), cada um dos quais requer alguns recursos, limitando, portanto, a escalabilidade da solução.

Para evitar esta situação, poderão empregar a classe *java.util.concurrent.ScheduledThreadPoolExecutor*, a qual permite o agendamento de *timeouts*

7. Eliminar qualquer possibilidade de Bloqueio

Para uma escalabilidade máxima, deve ser eliminado o recurso a qualquer função bloqueante, nomeadamente, as chamadas de acesso ao sistema de ficheiros. Para isso poderão usar a classe *java.nio.channels.AsynchronousFileChannel*.

8. Conclusão

Este documento foca-se nos aspectos de concorrência do Projeto 1 e assume implicitamente que o ficheiro alvo do *backup* tem apenas um *chunk*. As sugestões antes apresentadas podem, de qualquer das formas, ser aplicadas ao *backup* de ficheiros com mais de um *chunk*, bem como à implementação dos outros subprotocolos.

Devem seguir uma abordagem incremental. No entanto, neste caso, vocês tem um "espaço de soluções" que é, pelo menos, bidimensional: numa das dimensões, está o nível de simultaneidade e escalabilidade pretendido para a vossa solução e, na outra dimensão, estão os subprotocolos (o número de *chunks* por arquivo adiciona, de certa forma, uma terceira dimensão).

A melhor abordagem será, provavelmente, uma do tipo "depth-first", na qual vocês primeiro implementam o *Chunk Backup Protocol* com o nível de simultaneidade que desejarem. Depois implementam os outros protocolos com o mesmo nível de simultaneidade. Devem, no entanto, gerir o vosso tempo: se demorarem muito para tratar da simultaneidade, depois talvez vos falte tempo para implementar os restantes protocolos.

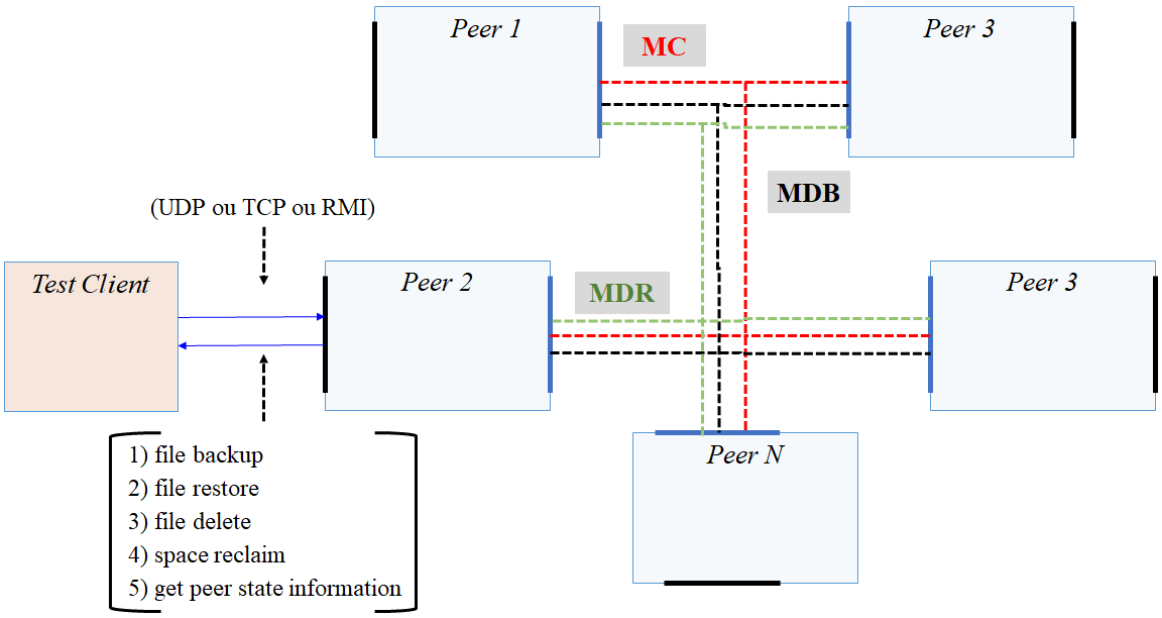
9. Leitura Adicional

Existem certamente muitos recursos bibliográficos relativos ao assunto que aqui abordamos. Os seguintes tutoriais parecem-me, no entanto, apresentar um bom equilíbrio entre teoria e prática e, portanto, podem ser úteis ao desenho e implementação do vosso primeiro projeto de SDIS (em Inglês):

- Java Concurrency/Multithreading Tutorial – A tutorial discussing concurrency issues in the context of Java. You may wish to take a look at the Concurrency Models chapter, which discusses several concurrency designs more abstractly than these notes.
- Java Concurrency Utilities – A tutorial on the *java.util.concurrent* package, including *ThreadPoolExecutor* and *ScheduledExecutorService*, which is the interface provided by the class *ScheduledThreadPoolExecutor*.
- Java NIO Tutorial – A good tutorial on Java NIO, including the *java.nio.channels.AsynchronousFileChannel*.

Notas sobre Trabalho 1

Já está disponível na página da cadeira o enunciado do primeiro projecto que vocês terão de desenvolver para avaliação.



Descrição sumária da estrutura e funcionamento do sistema a implementar:

- O trabalho a desenvolver constitui uma arquitectura P2P dedicada ao *backup* distribuido de ficheiros. Existirá assim um colectivo de *peers* que cooperará para armazenar de forma distribuida os ficheiros especificados, de acordo com um grau de replicação (número de cópias) também especificado. O sistema suportará também os serviços associados de *recovery*, *deletion* e *space reclaiming*;

- A arquitectura em questão compreende os dois seguintes tipos de componentes:
  - o *peer* – aplicação *p2p* que tanto recebe pedidos de outros *peers* como submete pedidos a outros *peers*. O sistema será assim composto por vários peers *idênticos* a correr simultaneamente e a interagir;
  - o *testClient* – aplicação que efectuará o interface entre o utilizador e um determinado *peer*. Apenas 1 *testClient* será empregue para intermediar a interacção entre o utilizador e um determinado peer;

- A comunicação entre os diferentes componentes proceder-se-á da seguinte forma:
  - o *testClient* comunica com o *peer* através (preferivelmente) de RMI;
  - os *peers* comunicam uns com os outros através de 3 canais multicast.

- Os canais multicast serão os seguintes:
  - MC – *Multicast Control Channel*. Canal empregue para a troca das mensagens de controlo do sistema. Todos os peers devem *estar à escuta* deste canal;
  - MDB – *Multicast Data Backup Channel*. Canal empregue nos procedimentos de *backup*;
  - MDR – *Multicast Data Recovery Channel*. Canal empregue nos procedimentos de *recovery*;

- Sobre a arquitectura descrita deverão ser suportados 4 serviços (protocolos) relacionados (oferecidos pelo *testClient* ao utilizador):
  - *Backup* – para replicar um ficheiro com um grau de replicação especificado;

  - Recovery – para recuperar um ficheiro que foi previamente replicado;
  - *Delete* – para eliminar do sistema um ficheiro antes replicado;
  - *Space Reclaiming* – para libertar espaço (disco) de um *peer* garantindo a preservação do *replication degree* dos ficheiros (*chunks*, para ser mais correcto) por ele armazenado;

- A operação do sistema será a seguinte:
  - o utilizador solicita um dos serviços suportados ao *testClient*;
  - o *testClient* constrói e envia ao *peer* o pedido correspondente;
  - o *peer* que recebe o pedido (designado como *initiator peer*) interage com os restantes *peers* (através dos canais *multicast* adequados) para levar a cabo os necessários procedimentos para assegurar o serviço solicitado (*backup*, *recovery*, *delete* ou *space reclaim*). Aos procedimentos desencadeados para suportar um determinado serviço (ou protocolo) chama-se (no guião) um sub-protocolo;
  - notem que o suporte de um determiando serviço pode despoletar um vasto conjunto de interacções envolvendo outros *peers* que não o próprio *initiator peer*;

- É de notar que (no contexto das operações/interacções que se desenvolvem para dar suporte aos serviços oferecidos pelo sistema) os *peers* não operarão sobre os ficheiros inteiros, mas sobre fragmentos destes, designados de *chunks*.
  Assim, aquando do *backup*, o ficheiro inteiro será fornecido ao *testClient* (pelo utilizador), que o fará chegar ao seu *peer*. Nesse ponto cada *peer* deverá fragmentar cada ficheiro no numero necessário de *chunks* para que cada um destes possa ser transportado no *payload* de um datagrama UDP (para poder circular nos canais de comunicação entre peers).
  Depois o *peer* deverá desenvolver o mesmo procedimento para cada *chunk* (*backup*). Estes procedimentos deverão ser executados concorrentemente, usando múltiplos *threads*.

- Da mesma forma, a recuperação (*recovery*) de um ficheiro implicará a recuperação individual de dada um dos *chunks* em que este foi dividido aquando do *backup*. O *initiator peer* tratará de obter (pedir aos outros *peers*) todos os *chunks* de um ficheiro, reconstituirá o ficheiro e só depois o devolverá ao *testClient* (e, portanto, ao utilizador);

- Em face do que é expresso acima fica claro que o *peer* deverá estar à escuta de pedidos vindos de dois lados: do *testClient* (através de uma ligação RMI) e do colectivo de *peers* (através dos 3 canais multicast). Um peer deve assim apresentar dois interfaces:
  - o interface para o *testClient* – serviço que permita ao *testClient* a submissão de pedidos de *backup*, *recovery*, *deletion*, ou *space reclaim* (preferencialmente usando Java RMI (vale 5%), mas pode no entanto ser implementado de outra forma, p.ex. usando TCP ou UDP);
  - o interface para os restantes peers – aqui referimo-nos aos mecanisos do *peer* que estão à escuta dos pedidos provenientes dos 3 canais multicast, portanto, à escuta de pedidos dos outros peers;

A acção dentro de um *peer* pode portanto ser desencadeada por um pedido vindo do *testClient* (utilizador) ou de um outro *peer*.

Notas gerais:

- Como não é possível testar os restantes protocolos/serviços sem implementar o protocolo de *Backup*, o recomendável é que comecem por implementar este protocolo. Para além disso, embora se pretenda uma implementação concorrente baseada em múltiplos *threads*, inicialmente podem fazer uma implementação mais simples, capaz de fazer o *backup* de um ficheiro com um *chunk* apenas;
- A ordem gobal mais adequada para implementarem os sub-protocolos é: *backup*, *delete*, *restore* e *reclaim*;

Notas sobre Trabalho 1

- Devem fazer uma implementação incremental. Ou seja implementar cada funcionalidade necessária isoladamente (p.ex. enviar/receber mensagens em multicast, calcular o identificador do ficheiro para *backup*, partir o ficheiro em *chunks*, etc.) e ir integrando essas funcionalidades gradualmente;
- Há um conjunto de possíveis *enhancements*, referidos ao longo do enunciado, que vocês podem implementar. Devem é implementar essas melhorias depois de completarem as versões base de todos os protocolos.
- A demonstração do vosso projecto requererá que seja possível executar os vossos *peers*, assim como o *testClient*, a partir da linha de comandos. Recomendamos assim que vocês criem um script para agilizar esse processo;
- É também necessário, para efeitos de teste, que a vossa implementação permita executar múltiplos *peers* no mesmo computador;
- Para além disso, a demonstração será realizada em PCs da sala de aula, assim para não serem penalizados por problemas súbitos que surjam durante a demonstração ensaiem o *setup* do vosso sistema antes da demonstração;
- Notem que a montagem do vosso sistema para demonstração valerá 5% da nota do projecto;
- Um dos testes a que os vossos *peers* serão sujeitos é um teste de interoperabilidade com os *peers* de outros grupos. Assim, podem e devem realizar esses testes entre as vossas implementações. No entanto, para o fazerem podem partilhar apenas ficheiros *.class* (pré-binários java) com outros grupos e não ficheiros *.java* (código fonte).
- Relativamente ao *replication degree* é de notar que o valor especificado (aquando do *backup*) é apenas o valor desejado. Um *chunk* pode acabar por ser replicado com um grau inferior, se p.ex. não houver um número suficiente de *peers*, ou superior.

RPC: Remote Procedure Call

March 2, 2020

Roadmap

Idea

Implementation

Transparency

RPC Semantics in the Presence of Faults

Further Reading

Roadmap

Idea

Implementation

Transparency

RPC Semantics in the Presence of Faults

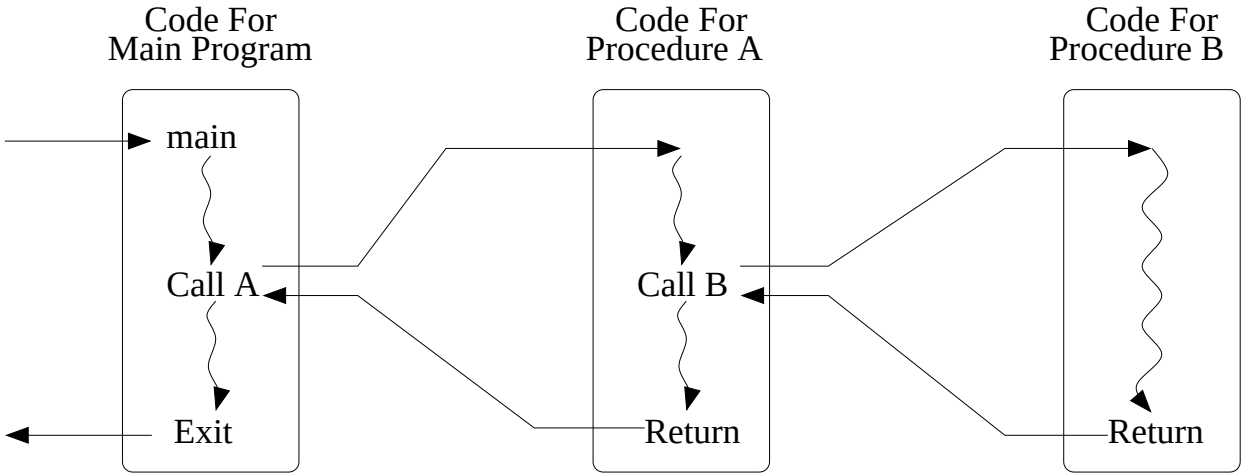
Further Reading

Remote Procedure Call (RPC)

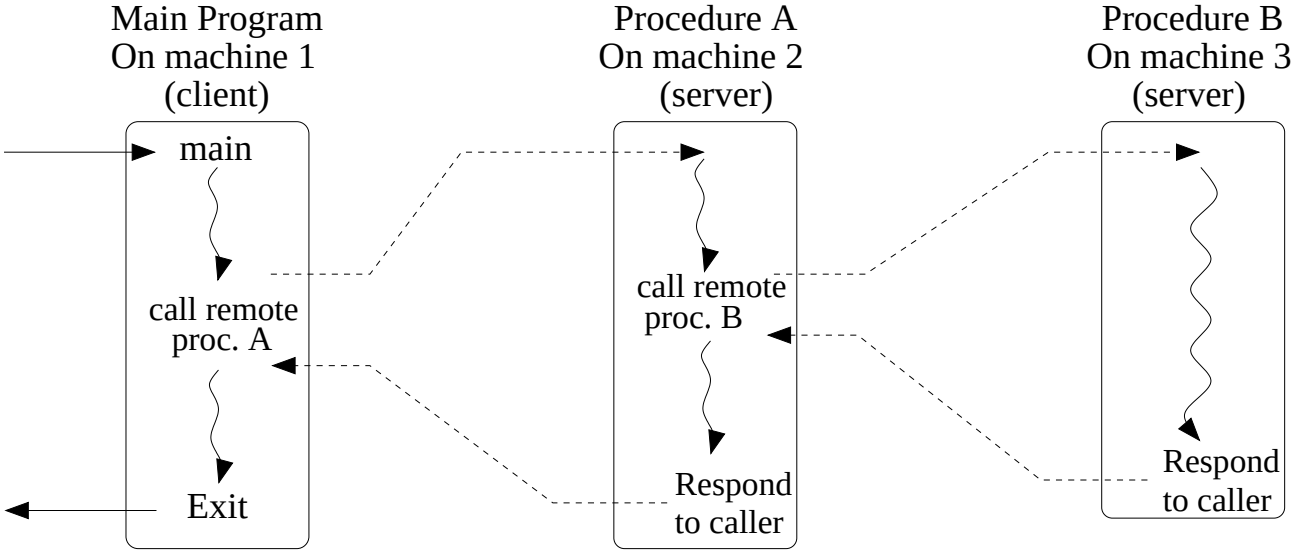
- Message-based programming with `send()/receive()` primitives is not convenient
  - depends on the communication protocol used (TCP vs. UDP)
  - requires the specification of an application protocol
  - akin to I/O
- Function/procedure call in a remote computer
  - is a familiar paradigm
  - eases transparency
  - is particularly suited for client-server applications

RPC: the Idea

Local procedure call:

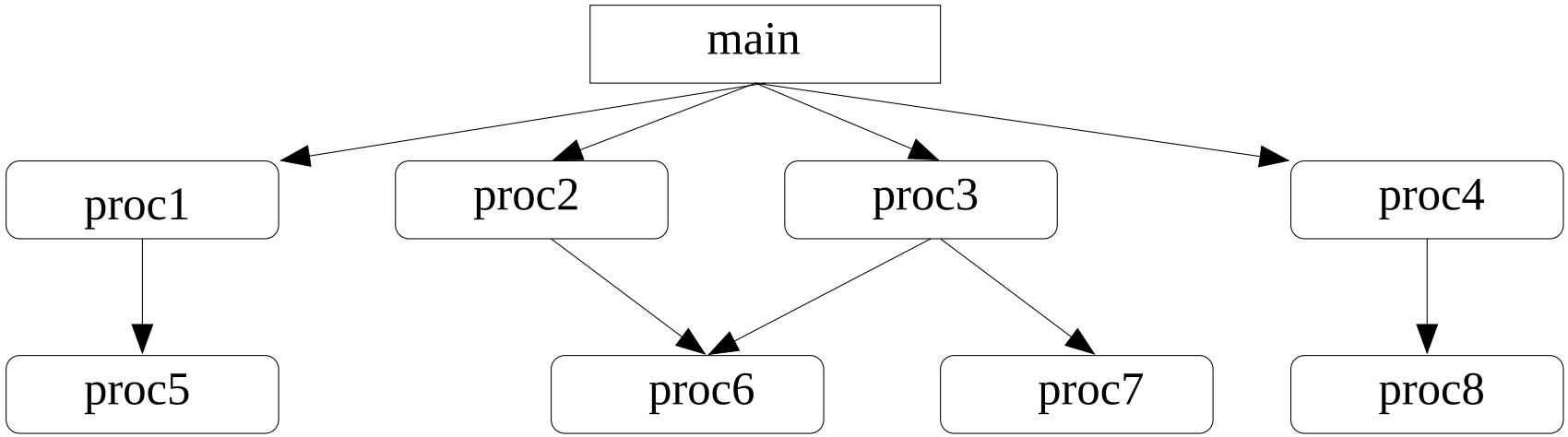


Remote procedure call:

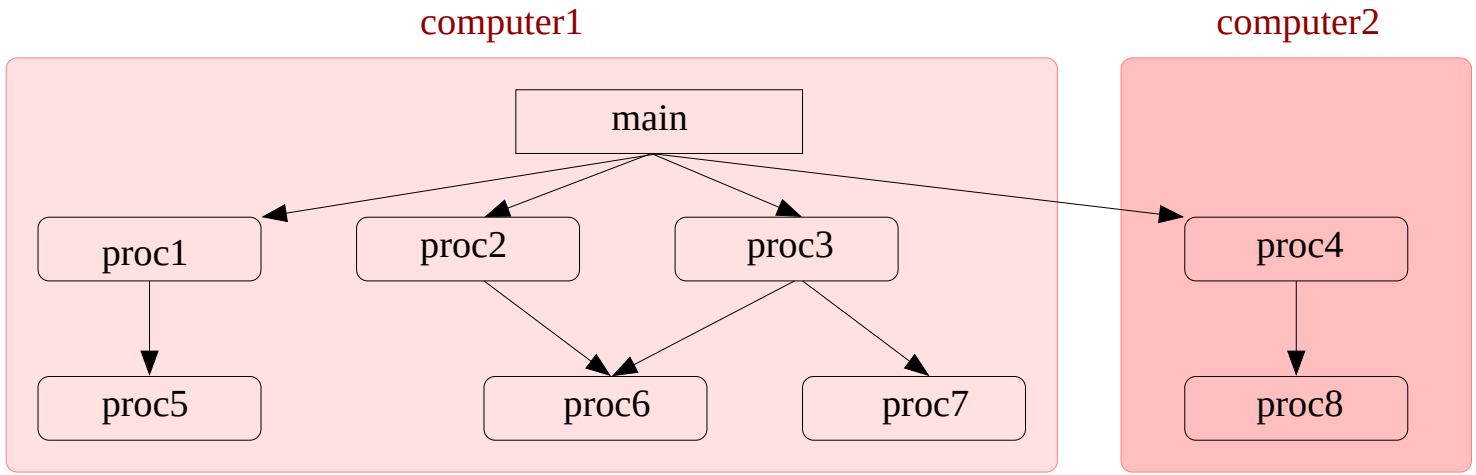


Program Development with RPCs: the Vision

- Design/develop an application ignoring distribution



- Distribute *a posteriori*



Roadmap

Idea

Implementation

Transparency

RPC Semantics in the Presence of Faults

Further Reading

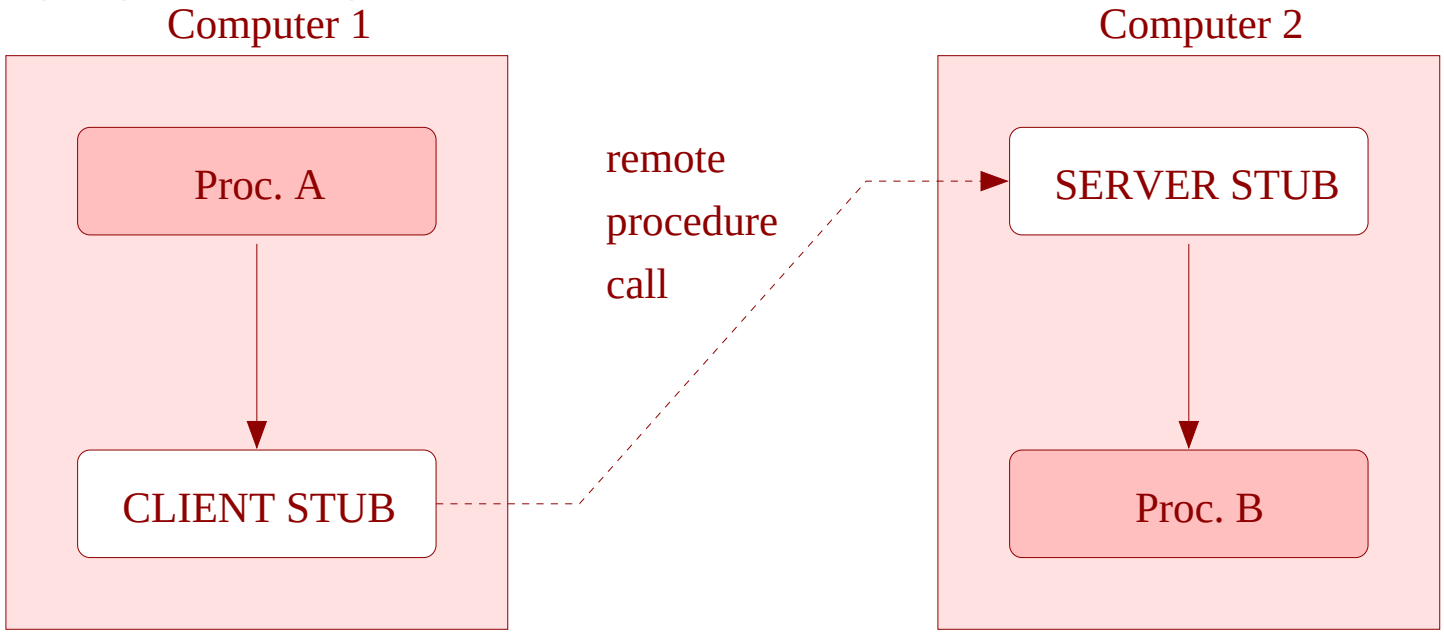
RPC Stub Routines

- Ensure RPC transparency

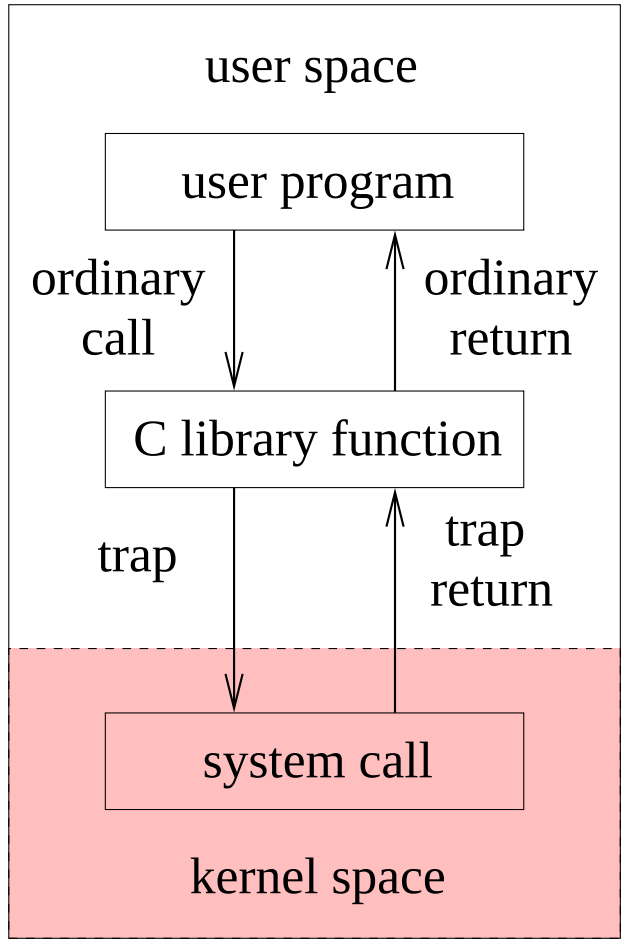
  Client invokes the **client stub** – a local function

  Remote function is invoked by the **server stub** – a local function

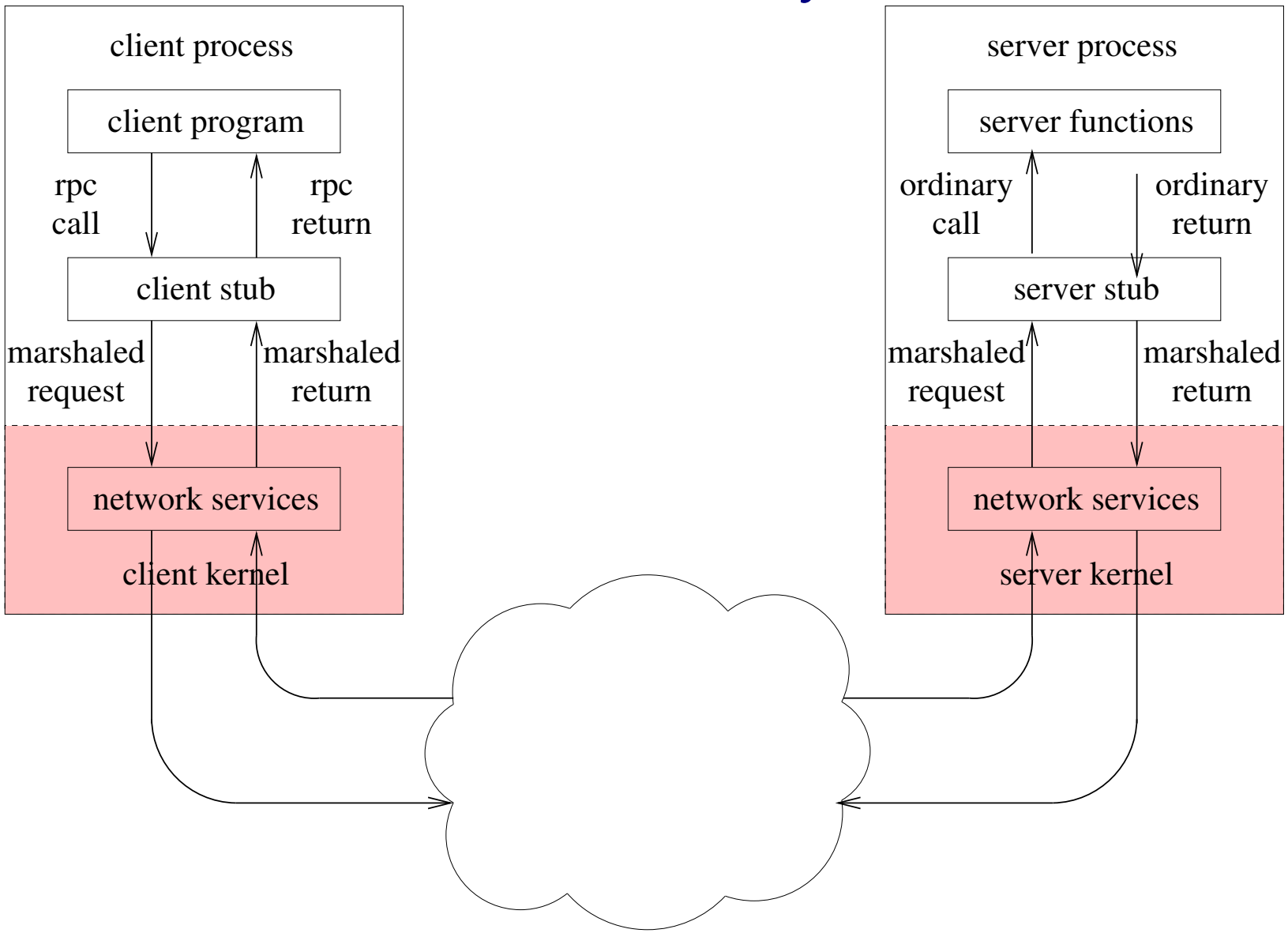
- The stub routines communicate with one another by exchanging messages



Well Known Trick: also Used for System Calls



Typical Architecture of an RPC System



Obs. RPC is typically implemented on top of the transport layer (TCP/IP)

Client Stub

Request

1. Assembles message: **parameter marshalling**
2. Sends message, via `write()/sendto()` to server
3. Blocks waiting for response, via `read()/recvfrom()`
   - Not in the case of **asynchronous RPC**

Response

1. Receives responses
2. Extracts the results (**unmarshalling**
3. Returns to client
   - Assuming **synchronous RPC**

Server Stub

Request

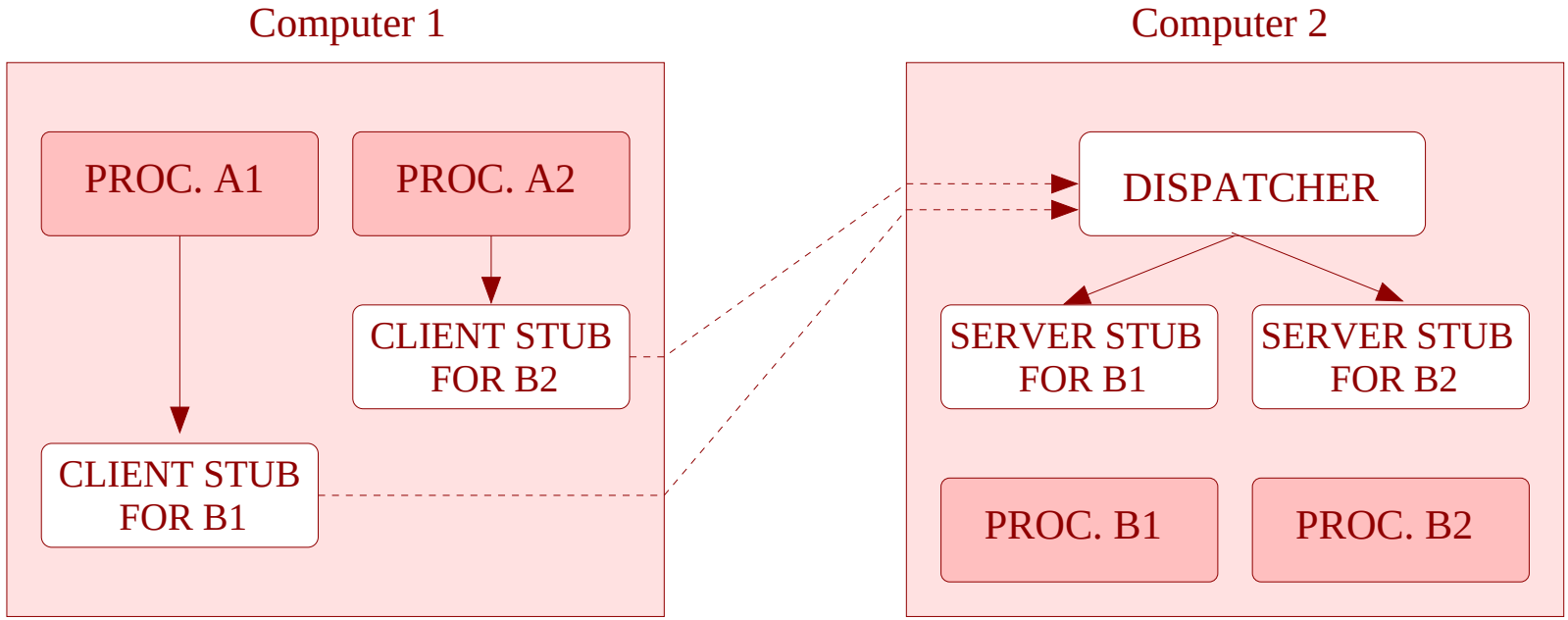
1. Receives message with request, via `read()/recvfrom()`
2. Parses message to determine arguments (**unmarshalling**
3. Calls function

Response

1. Assembles message with the return value of the function
2. Sends message, via `write()/sendto()`
3. Blocks waiting for a new request

RPC: Dispatching

- Often, **RPC services** offer more than one remote procedure:



- The identification of the procedure is performed by the **dispatcher**
  - This leads to a hierarchical name space **(service, procedure)**

Roadmap

Idea

Implementation

Transparency

RPC Semantics in the Presence of Faults

Further Reading

Transparency: Platform Heterogeneity

Problems at least two:

1. Different architectures use different formats
   - 1's-complement vs. 2's complement
   - big-endian vs. little-endian
   - ASCII vs. UTF-??
2. Languages or compilers may use different representations for composite data-structures

Solution mainly two:

standardize format in the wires

- \+ needs only two conversions in each platform
- – may not be efficient

receiver-makes-right

Transparency: Addresses as Arguments

Issue The meaning of an address (C pointer) is specific to a process

Solution Use **call-by-copy/restore** for parameter passing

- \+ Works in most cases
- – Complex
  - The same address may be passed in different arguments
- – Inefficient
  - For complex data structures, e.g. trees

Transparency in the Presence of Faults

Problem What if something breaks?

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?
- Client crashes

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?
- Client crashes
  - Need to prevent **orphan** computations, i.e. on behalf of a dead process.

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?
- Client crashes
  - Need to prevent **orphan** computations, i.e. on behalf of a dead process.

Issue A client cannot distinguish between loss of a request, loss of a response or a server crash

- The absence of a response may be caused by a slow network/server

Roadmap

Idea

Implementation

Transparency

RPC Semantics in the Presence of Faults

Further Reading

RPC Semantics in the Presence of Faults (Spector82)

Question What can a client expect when there is a fault?

Answer Depends on the semantics in the presence of faults provided by the RPC system

At-least-once Client stub must keep retransmitting until it obtains a response

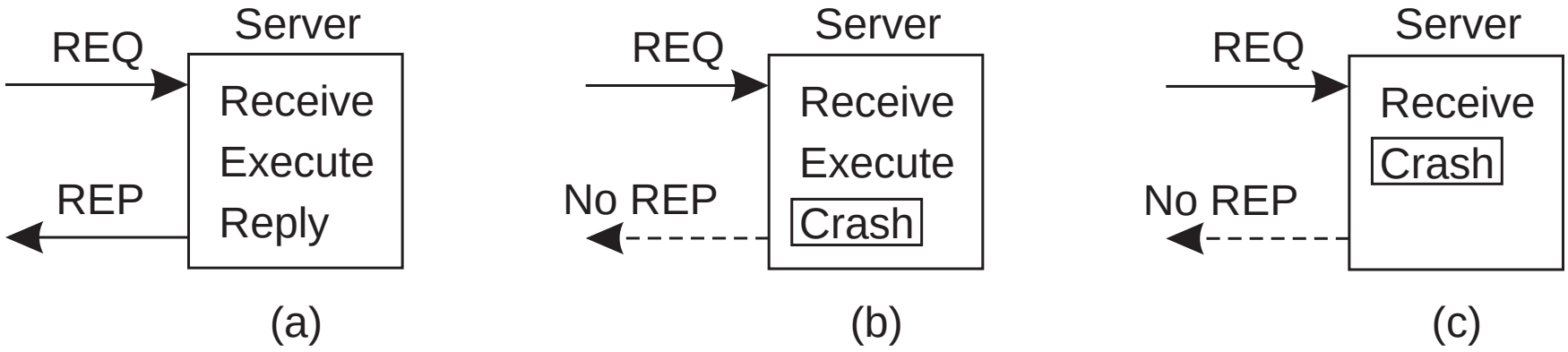
- Be careful with non-idempotent operations
- Spector allows for zero executions in case of server failure

At-most-once Not trivial if you use a non-reliable transport, e.g. UDP.

- If the RPC uses TCP, it may report an error when the TCP connection breaks

Exactly-once Not always possible to ensure this semantics, especially if there are external actions that cannot be undone

Faults and Exactly-once Semantics



Problem In the case of external actions, e.g. file printing, it is virtually impossible to ensure Exactly-once Semantics

Server policy One of two:

1. Send an `ACK` after printing
2. Send an `ACK` before printing

Client policy One of four:

1. Never resend the request
2. Always resend the request
3. Resend the request when it receives an `ACK`
4. Resend the request when it does not receive an `ACK`

Server Faults and Exactly-once Semantics

Scenario Server crashes and quickly recovers so that it is able to handle client retransmission, but **it has lost all state**

Let

`A`: ACK `P`: print `C`: crash

Fault scenarios (`ACK->P`)

1. `A->P->C`
2. `A->C(->P)`
3. `C(->A->P)`

Fault scenarios (`P->ACK`)

1. `P->A->C`
2. `P->C(->A)`
3. `C(->P->A)`

Client

Reissue Strategy
Always
Never
When Ack
When not Ack

OK = Text printed once

Server

Strategy A→P		
APC	**AC(P)**	**C(AP)**
Dup	OK	OK
OK	Zero	Zero
Dup	OK	Zero
OK	Zero	OK

Dup = Text printed twice

Strategy P→A		
PAC	**PC(A)**	**C(PA)**
Dup	Dup	OK
OK	OK	Zero
Dup	OK	Zero
OK	Dup	OK

Zero = Text not printed at all

Conclusion No combined strategy works on every fault scenario

- What if server saved state on disk?

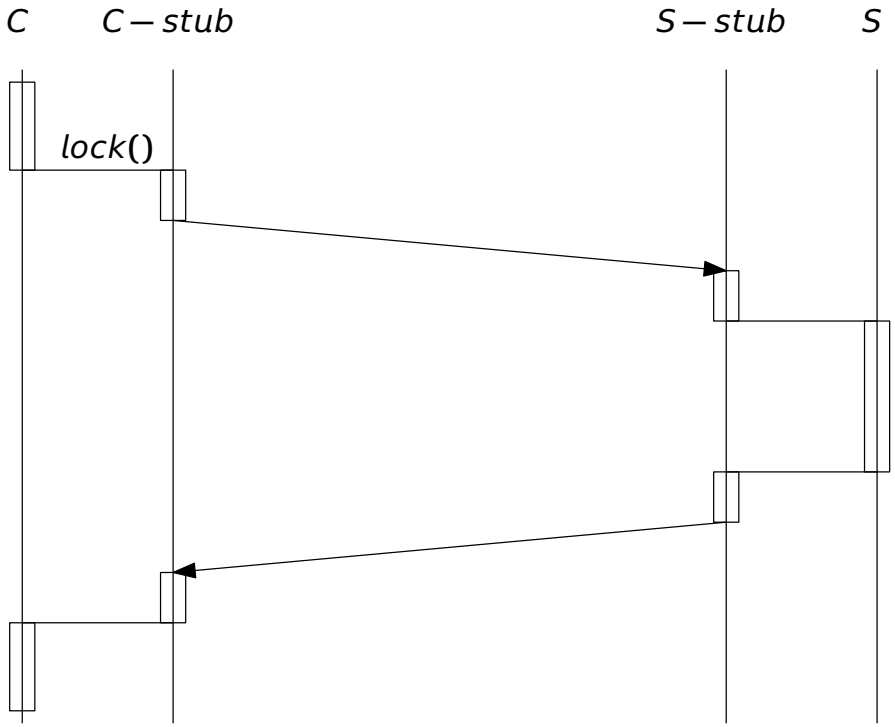
At-least-once vs. At-most-once

- Consider a locking service using two RPCs:

  `lock()`

  `unlock()`

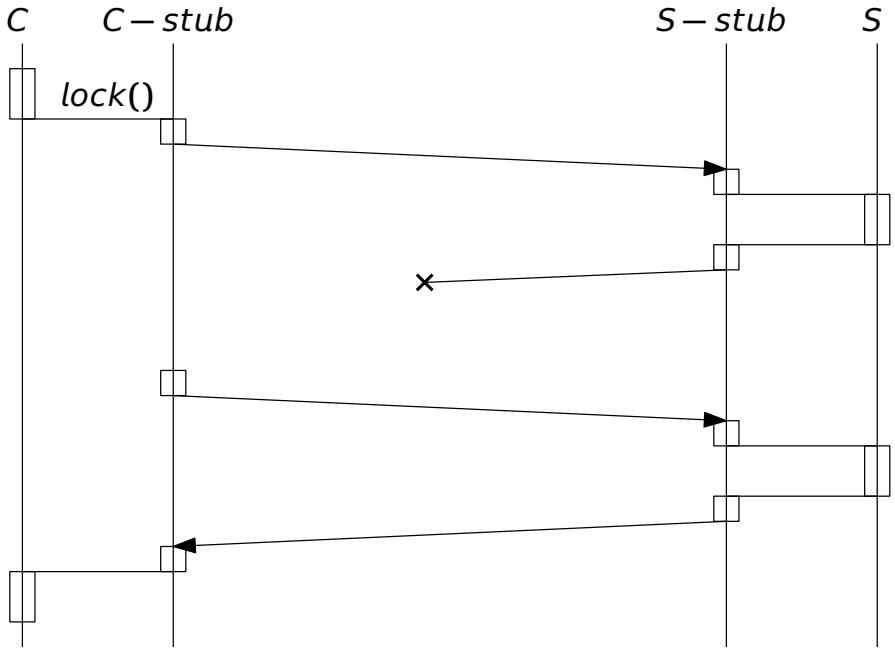
No failures and no message loss



- It does not matter the semantics supported by the RPC library

At-least-once vs. At-most-once: Lost Response

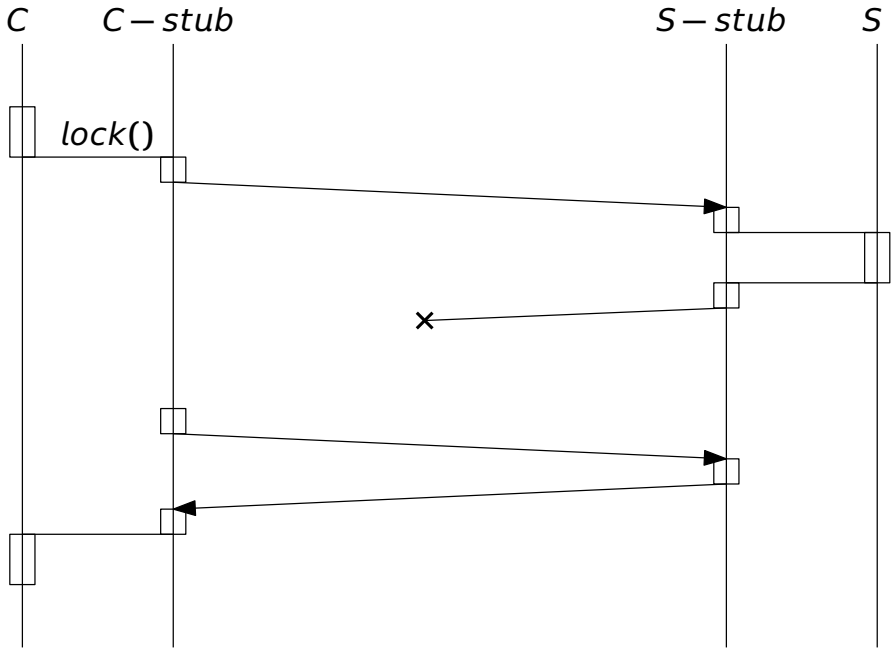
At-least-once



- Remote procedure may be invoked more than once
  - If procedure is not **idempotent**:
    - RPC must include an id as argument
    - Server must keep table with responses previously sent
  - Is `lock()` an idempotent procedure?

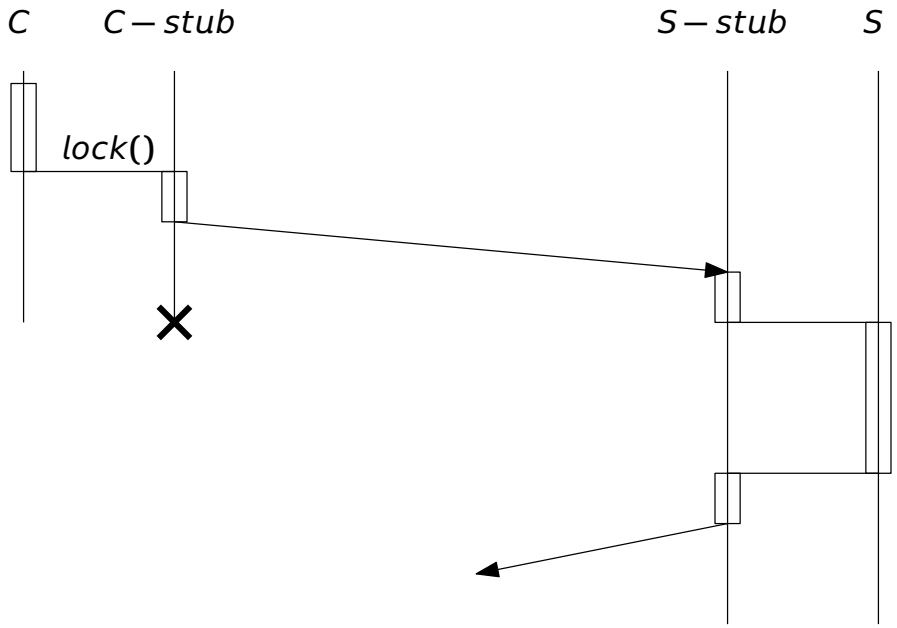
At-least-once vs. At-most-once: Lost Response

At-most-once (UDP?):



- There is no guarantee that the procedure will be executed
  - But in that case, the caller should receive an exception
- The RPC middleware ensures that the procedure is not executed more than once
  - RPC requests include an id
  - RPC system keeps table with responses
- What would be different if using TCP?

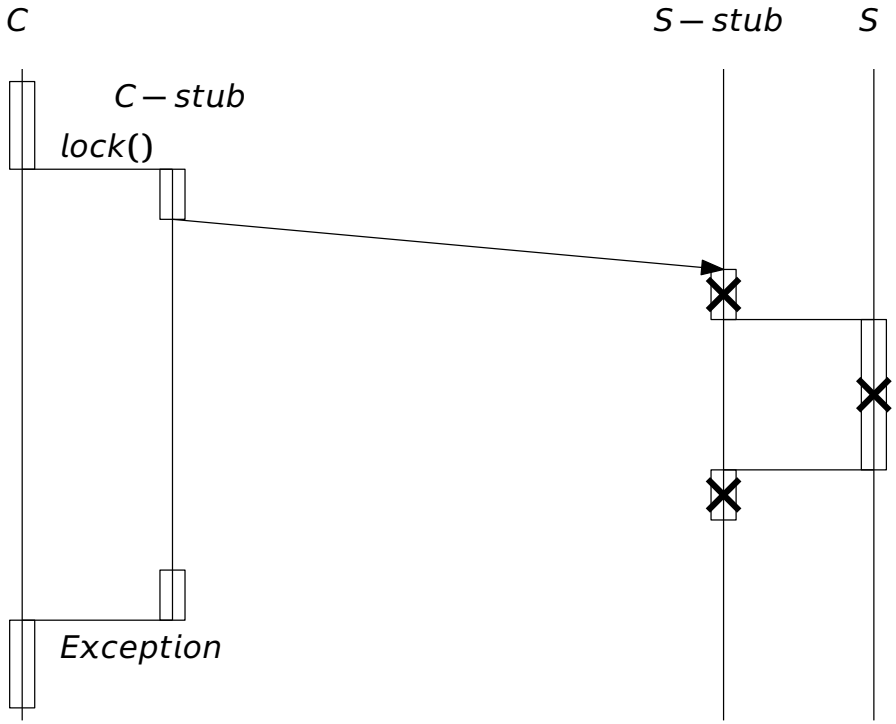
At-least-once vs. At-most-once: Client crash



- Again, the RPC semantics is irrelevant

At-least-once vs. At-most-once: Server crash

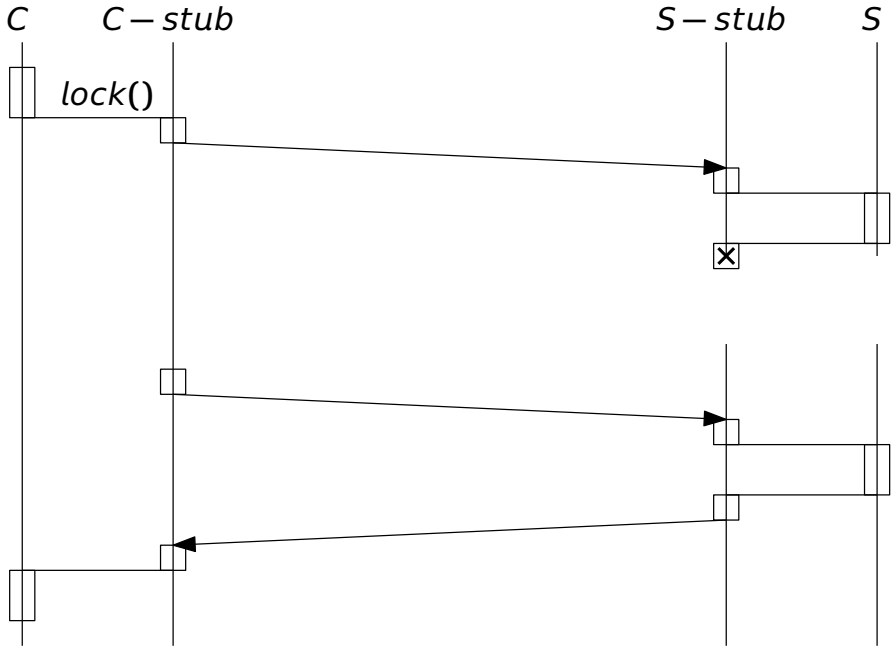
At-most-once



- Client does not know if server granted it the lock
  - Depends on when the server crashed
- Client, **not RPC**, may ask the server (or just retry)
  - Server needs to remember state across reboots
    - E.g. store locks state on disk
- Is this different from an exception upon message loss?

At-least-once vs. At-most-once: Server crash

At-least-once



- Server may run the procedure several times
  - Client stub may send several requests before giving up
- Server needs to remember previous requests across reboots (if requests are not **idempotent**). E.g.:
  - Store table request ids on disk
  - Check the request table on each request

At-least-once vs. At-most-once: Conclusions

Message loss

At-least-once

- Suits if requests are idempotent

At-most-once

- Appropriate when requests are not idempotent

Server crashes

- No clear advantage: the service itself may have to take special measures

Upon an exception can the caller tell whether the cause is message loss or server crash?

Roadmap

Further Reading

- Tanenbaum e van Steen, *Distributed Systems, 2nd Ed.*
  - Section 4.2 *Remote Procedure Call*, except subsection 4.2.4
  - Subsection 8.3.2 *RPC Semantics in the Presence of Failures*
- Birrel and Nelson, *"Implementing Remote Procedure Calls"*, ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, Pages 39-59
- A. Spector, *"Performing Remote Operations Efficiently on a Local Computer Network"*, Communications of the ACM, Vol. 25, No. 4, April 1982, Pages 246-260

1. Introdução

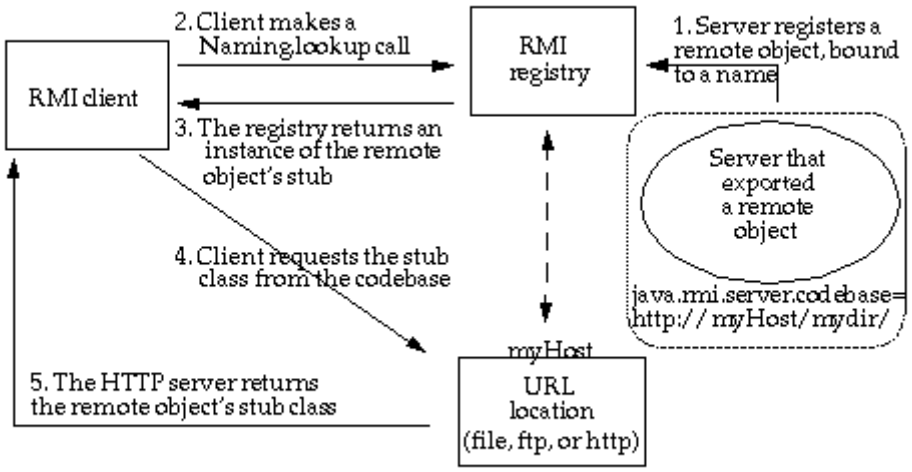
Na aula TP de SDist desta semana, trabalharemos no exercício apresentado em [1]. O principal objetivo desta aula é que vocês entendam os conceitos de base da comunicação sobre Java RMI.

No final de [1] encontram um tutorial útil sobre comunicação Java RMI (este [2]).

Tanto o exercício desta aula como o exemplo do tutorial Oracle [2], concentram-se no desenvolvimento de uma interface RMI entre um cliente e um servidor que poderá depois, de alguma forma, ser reutilizada no primeiro projecto avaliado.

(especificamente na comunicação entre o *TestClient* e o *Peer*, caso pretendam implementar essa comunicação usando o RMI).

2. Comunicação RMI



Na comunicação RMI, a interacção entre um servidor e um cliente é assistida (numa fase inicial) por outra aplicação, i.e. o *rmiregistry*.

O *rmiregistry* é um serviço de registo de nomes (nomes de outros serviços) para *bootstrap* da comunicação entre clientes e servidores. O *rmiregistry* regista assim uma associação entre nomes de serviços e os *remote objects* que os implementam, mais especificamente entre nomes de serviços e as referências para esses *remote objects* (designadas como os *stubs* desses *remote objects*).
Os servidores procedem ao registo dessas associações, i.e. inscrevem no *rmiregistry* pares de *nome-de-serviço/stub*.
Os clientes, tanto no *host* local como em *hosts* remotos, contactam o *rmiregistry* para obterem os *stubs* de serviços específicos, que depois empregam para invocar tais serviços.

No entanto, os *stubs* (ou as referências para *remote objects*) registados no *rmiregistry* não são suficientes para que o cliente possa aceder ao servidor. O cliente precisa dos ficheiros *.class* com a definição da classe do *stub* (mais especificamente a classe que implementa o serviço remoto) e das restantes classes de que esta necessite.

No contexto de Java RMI, o local/serviço onde estes ficheiros *.class* estão armazenados, para serem disponibilizados ao lado cliente, chama-se *codebase* (a qual é independente do *rmiregistry*)

A *codebase* pode ser assim definida como a fonte a partir da qual os ficheiros *.class*, com as implementações dos artefactos de comunicação, podem ser carregados para uma máquina virtual. Para obterem uma informação mais detalhada sobre a *codebase* e sobre como o lado do servidor deve lidar com ela, podem consultar a página indicada em [3].

A operação básica da comunicação em RMI Java é a seguinte (de [3]):

1. O servidor registra um *remote object*, associando-o a um nome, no *rmiregistry*. O que é registado no *rmiregistry* não é o efectivo *remote object*, é apenas uma referência para um *remote object* (ou *stub*). Esse *stub* é gerado de forma dinâmica (programaticamente) no servidor.
   Para que o *stub* possa ser executado no lado cliente é necessário que este consiga obter a informação em falta no *stub* (os ficheiros *.class* com as definições das classes que implementam o serviço remoto) que está armazenada na *codebase*. O que acontece em Java RMI é que a localização dessa *codebase*, é anotada no *stub* quando este é produzido (é daí que o cliente a obtém).
   Assim aquando de construção dinâmica do *stub* pelo lado servidor, a sua VM tem de saber onde está a *codebase*. Para isso esta propriedade, da VM em questão, deve ser previamente estabelecida (configurando a propriedade *java.rmi.server.codebase* na linha de comando ou programaticamente).
   Só depois de estar configurada a referida propriedade, e gerado o *stub* é que se pode proceder ao registo efectivo.

2. O cliente solicita, ao *rmiregistry*, uma referência para um *remote object* específico.

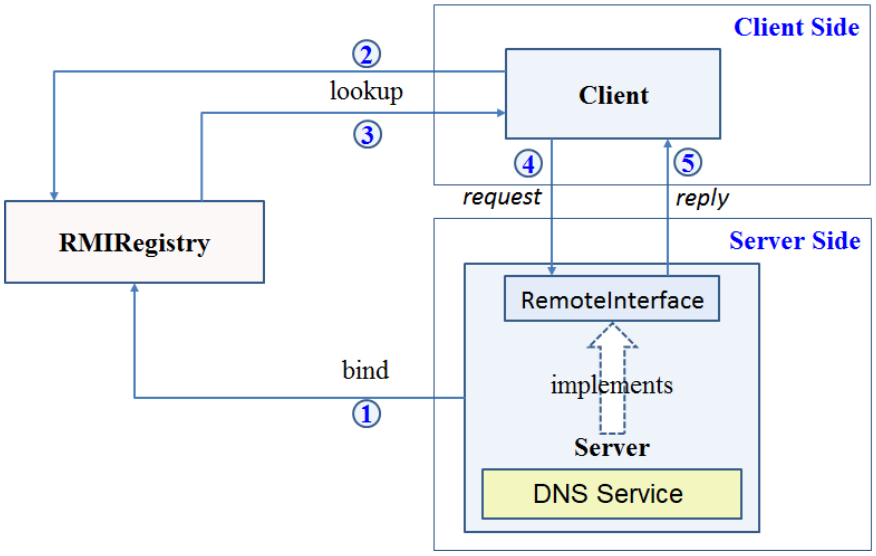
3. O *rmiregistry* retorna a referência (*stub*) em questão.

4. O cliente trata de obter a(s) definição(s) da(s) classe(s) da necessárias à execução do *stub*. Se estas definições (*.class*) puderem ser obtidas da *classpath* do cliente (que é sempre pesquisada antes da *codebase*), o cliente carregará a classe localmente. Caso contrário o cliente tentará obter as definições de classes da *codebase*, empregando o valor da *codebase* (URL) que está anotado no próprio *stub*.

5. A definição de classe para o *stub* (e quaisquer outras classes necessárias) é retornada ao cliente.

6. Neste ponto, o cliente tem todas as informações necessárias para invocar os métodos oferecidos pelo *remote object*. A instância do *stub* funciona assim como um *proxy* para o *remote object* que executa do lado servidor. Desta maneira, e diferentemente de um *applet* que recorre a uma *codebase* para obter código remoto e executa-lo na sua VM local, o cliente Java RMI recorre a uma *codebase* para obter código remoto que desencadeia a execução de código noutra VM potencialmente remota.

3. Exercício do Lab 3



A imagem acima apresenta uma descrição simples da arquitetura que deverão implementar no lab 3 (bem como no tutorial da Oracle [2]). É muito próximo daquilo que já foi apresentado na imagem anterior, e assim, também é a sua explicação.

As etapas para desenvolver uma aplicação cliente-servidor utilizando Java RMI são as seguintes:

1. Definir a interface remota oferecida pelo servidor. Terão assim de definir um interface que estenda o interface *java.rmi.Remote*. Estender este interface implica as seguintes restrições:
   a) todos os métodos declarados (no interface do serviço) devem lançar uma excepção do tipo *java.rmi.RemoteException* (essa excepção é lançada pela JVM no caso de problemas de comunicação entre as JVMs);
   b) os argumentos e valores de retorno de todos os métodos declarados devem implementar a interface *java.io.Serializable*.

2. Desenvolver a classe do servidor, ou seja, a classe que implementa a interface definida no ponto anterior. Esta classe (ou a classe que inicia a execução do servidor) deve efectuar os seguintes passos:
   a) fazer, programaticamente, o set da propriedade *java.rmi.server.codebase* com o valor da *codebase* (para que este valor seja conhecido pela VM e possa ser anotado no *stub* aquando da sua criação);
   b) instanciação do "*remote object*". Para isso o primeiro passo é criar uma instância da classe servidor (usando o seu constructor). Depois é necessário "exportar" o *remote object*, ou seja, colocar o serviço a correr numa determinada porta e produzir o respectivo *stub*. As duas tarefas são realizadas pelo método método *static export(….)* da classe *java.rmi.server.UnicastRemoteObject.* O valor de retorno deste método é o *stub*;
   c) registo do *stub*, associado a um nome, no *rmiregistry*. Para isso existem duas opções:
      a. empregar os métodos *static bind(..)* ou *rebind(…)* da classe *java.rmi.Naming*. Estes permitem efectuar o registro de uma interface remota no *rmiregistry*;
      b. empregar o método *getRegistry()* da classe *java.rmi.registry.LocateRegistry*. Obtém-se assim uma referência a um objecto que implementa o interface *java.rmi.registry.Registry* (ou seja, retorna um objeto que representa o *rmiregistry*), a qual oferece os métodos descritos acima *bind()* e *rebind()*.

Em ambas as situações é preferível usar *rebind(…)*, pois *bind(…)* lançará uma excepção caso nome registrado anteriormente (ou seja, uma *String*) seja reutilizado. *rebind(…)* reassocia o nome ao "novo" *remote object*.

3. Desenvolver o cliente. Este deverá executar os seguintes passos:
   a) obter, do *rmiregistry*, o *stub* de acesso ao serviço remoto, ou seja, fazer o *lookup* do serviço. Para isso há duas alternativas (de forma semelhante ao que é descrito em 2c):
      a. empregar o método *static lookup(..)* da classe *java.rmi.Naming*. Este permite a obtenção de um *stub* para um *remote object* registrado no *rmiregistry*.
         Neste caso o argumento do método *lookup(…)* deve assumir o seguinte formato: //<host_name> [: < port_number >] / <obj_name> - em que < port_number > é o número da porta usada pelo *rmiregistry* (por defeito é 1099);
      b. empregar o método *getRegistry()* da classe *java.rmi.registry.LocateRegistry*. Obter assim uma referência a um objecto que implementa o interface *java.rmi.registry.Registry* e assim ter acesso ao método *lookup(…)*.
         Neste caso o argumento *String* do método *lookup(…)* pode ser apenas <obj_name>, pois os valores de <host_name> e de < port_number > podem ser especificados usando os argumentos transmitidos ao método *getRegistry()*;

      Ambos os métodos retornarão uma instância de um objecto que implementa o interface do serviço definido no ponto 1.
   b) após os passos anteriores o cliente estará então pronto para invocar os métodos remotos oferecidos pelo serviço de DNS de acordo com a assinatura dos mesmos. Essa invocação pode levar a uma excepção do tipo *java.rmi.RemoteException*, e deve, portanto, ser feita dentro de um bloco *try-catch*.

Notas relativas à execução do lab 3 (execução de rmiregistry, servidor e cliente):

1. O *rmiregistry* deve estar a correr antes de se executar o servidor. Portanto, primeiro corre-se o *rmiregistry*, depois o servidor e depois o cliente;

2. O *rmiregistry* pode ser executado a partir da linha de comando ou programaticamente. Recomenda-se a primeira alternativa;

3. Para poder executar o *rmiregistry* (a partir da linha de comando), o seu executável deve estar na *classpath* da *shell*. É necessário verificar se o diretório *bin* da instalação Java (onde se encontra o *rmiregistry.exe*) foi adicionado à variável de ambiente *Path* (esse é o nome da variável no Windows);

4. Para simplificar a execução do *rmiregistry* este deve ser executado no mesmo diretório em que o servidor é executado (assumindo que este é executado na base da árvore directorial que contém os *.class*).
   Executar o *rmiregistry* no diretório em questão é vantajoso pela seguinte razão: se nada for especificado, o *rmiregistry* assume que seu diretório de execução é também a *codebase*. Dessa maneira, se o *rmiregistry* for executado no mesmo diretório do servidor, o primeiro encontrará automaticamente os ficheiros *.class* com a implementação do serviço;

5. O *rmiregistry* também pode ser executado a partir de outro diretório. Nesse caso, a propriedade *java.rmi.server.codebase* deve ser configurada. Para isso, no momento da inicialização do *rmiregistry*, a codebase deve ser especificado (por exemplo, *rmiregistry -J-Djava.rmi.server.codebase=file:///a/b/SDist_2018/L03/bin/* onde file:///a/b/SDist_2018/L03/bin/ é um valor a adaptar às características da vossa implementação;

6. No que respeita à execução do servidor, caso a propriedade *java.rmi.server.codebase* não seja configurada programaticamente, isso deverá então ser feito na linha de comandos tal como é explicado no ponto 5:

7. Em relação ao cliente, poderá ser necessário configurar a política de segurança para que a VM carregue "executáveis" remotos. Para isso recomenda-se a consulta do ponto D da seção 6.0 da referência [3].

4. Referências

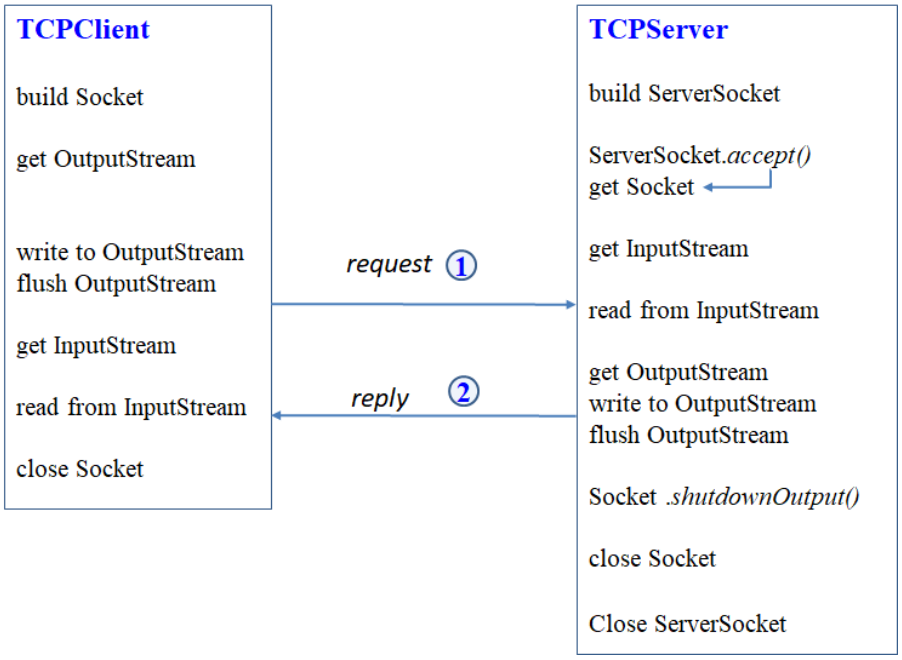
[1] Lab on RMI – https://web.fe.up.pt/~pfs/aulas/sd2020/labs/l03/rmi_l03.html

[2] Getting Started Using Java RMI, Oracle Tutorial, https://docs.oracle.com/javase/6/docs/technotes/guides/rmi/hello/hello-world.html

[3] Dynamic code downloading using Java RMI, Oracle, https://docs.oracle.com/javase/6/docs/technotes/guides/rmi/codebase.html

Exercício sobre TCP

1. Arquitectura Geral



2. Objectivos para a Aula 6

O objetivo da 6ª aula TP é permitir a vossa aquisição de conhecimentos no contexto da comunicação TCP (empregando a API Java para esse efeito).

O guião do exercício a ser implementado na aula está disponível em https://web.fe.up.pt/~pfs/aulas/sd2020/labs/l04/tcp_l04.html
(trata-se de mais uma versão do lab1, desta vez empregando TCP na comunicação entre cliente e servido)

No contexto do *assigment 1* a API Java para TCP poderá ser empregue:

- na comunicação entre o *TestClient* e os *Peers*, caso não consigam/queiram desenvolver esta interface em RMI;
- no *enhancement* to "*Chunk Restore Protocol*" (secção 3.3 do guião do *assigment 1*);

3. Notas Sobre a API Java para TCP

As notas abaixo estão de acordo com as transparências apresentadas pelo Professor Pedro Souto na aula teórica, e presentes primeiro "link" no final do guião.

1) A API de Java distingue entre:

- sockets usados para aceitar conexões – deve usar-se uma instância da classe *java.net.ServerSocket* para ficar permanentemente á escuta de ligações TCP numa determinada porta;
- sockets usados para transferir dados – são usadas instâncias da classe *java.net.Socket* para a efectiva leitura e envio de informação para o canal TCP

2) Embora a classe *Socket* seja usada para a troca de informação, esta não oferece métodos para o fazer diretamente. Fornece antes os métodos *getInputStream()* e *getOutputStream()*. Estes retornam referências para objetos (que implementam as classes abstratas *InputStream* e *OutputStream*), os quais permitem a efectiva troca de informação.

3) Os objectos retornados pelos métodos acima mencionados são bastante básicos e permitem apenas transferir *arrays* de bytes.
Pode-se, no entanto, usar estes *streams* básicos para construir *streams* com mais funcionalidades (classes da *package java.io.*). Podem assim criar-se *streams* capazes de efectuar a escrita/leitura de:
- texto formatado (*PrintWriter*/*BufferedReader*),
- tipos primitivos de Java (*DataOutputStream*/*DataInputStream*)
- objetos Java serializáveis (*ObjectOutputStream*/*ObjectInputStream*)

Notem que é importante que os *streams* (instâncias de classes da package *java.io.*) empregues para enviar mensagens numa das extremidades, e recebê-las na outra, sejam os correspondentes (os pares acima indicados). Assim, é pouco útil, por exemplo, usar um *ObjectOutputStream* para enviar dados, se na outra extremidade se usa um *BufferedReader*.

No exercício da aula desta semana fará sentido usarem o par *PrintWriter*/*BufferedReader*. O emprego destas classes permite estruturar as mensagens como linhas e troca-las como tal, usando os métodos *println()* e *readLine()* daquelas classes.

No caso *assignment*, o emprego de TCP será para a transferência de *chunks* (secção 3.3) e assim o uso das classes *PrintWriter* e *BufferedReader* não é adequado. Deverá antes ser suficiente o uso dos *streams* básicos, i.e. *OutputStream* e *InputStream*.

4) Algumas das classes que implementam *streams*, (e.g. *PrintWriter*), fazem *buffer* dos dados antes de os enviarem. Assim podem não enviar os dados que são "escritos" neles imediatamente para o socket, o que poderá levar a comportamentos inesperados. É por isto conveniente que se faça *flush* do *stream* onde se está a escrever (invocando o método *flush()*).
No caso da classe *PrintWriter* existe um construtor que permite configurá-lo de forma a que este faça *flush* automaticamente sempre que nele é escrita uma linha.

5) Para garantir que ao fechar uma conexão (método *close()* das classes *Socket*) não se perdem dados (devido, por exemplo, à questão abordada no ponto anterior), antes desse fecho efectivo deverá chamar-se o método *shutdownOutput()*, e só depois então o metido *close()*.

Projecto 1 – Notas Sobre Concorrência

1. Introdução

Este documento sugere uma possível via para uma implementação altamente concorrente e escalável do Projeto 1. Usa-se como exemplo a implementação do *Chunk Backup Protocol* (mensagem **PUTCHUNK**).

O que vos propomos é que desenvolvam uma implementação gradual do protocolo e, através de iterações sucessivas, melhorem a simultaneidade/concorrência e a escalabilidade da vossa implementação. O que é apresentado nas secções seguintes segue precisamente este método de desenvolvimento gradual.

Este documento não cobre, obviamente, todos as soluções possíveis.
Se se sentirem confiantes no vosso trabalho de implementação com *threads* podem saltar diretamente para a seção 3.

2. Implementação *Single Threaded*

Suposição: nesta implementação considera-se que é apenas necessário executar uma instância do protocolo de cada vez.

No final da implementação desta versão (simplificada) do protocolo, serão capazes de efectuar o *backup* de ficheiros com um único *chunk*, ou com vários deles, desde que tratem do *backup* de um *chunk* apenas depois de terminar o processo de *backup* do *chunk* anterior.

A ideia de base consiste em usar um único *thread* para cada um dos seguintes:
- *peer* iniciador
- *peer* não iniciadores

Simplificação: dado que esta implementação se trata apenas de uma versão preliminar, não há necessidade de medir com precisão o intervalo entre as retransmissões (da mensagem **PUTCHUNK**). Em vez de se medir o intervalo entre retransmissões, mede-se antes a duração do silêncio no meio (canais), ou seja, desde a transmissão do **PUTCHUNK** mais recente ou desde a recepção do **STORED** mais recente.

Nesta versão simplificada do protocolo pode fazer-se uso de *DatagramSocket.setSoTimeout()* e *Thread.sleep()* para medir a passagem do tempo (e implementar tempos de espera), e usar um único *thread*, tanto para o *peer* iniciador como para os outros *peers*.

3. Um *Thread* por Canal *Multicast*, um Protocolo de cada Vez

Para esta versão da implementação, continuamos a assumir que não há execuções simultâneas de diferentes instâncias de protocolo (*Chunk Backup Protocol*), ou seja, a qualquer momento, há no máximo uma instância de protocolo em execução.

A ideia de base da solução presente consiste em empregar um *thread* por canal *multicast* para tratar da recepção das mensagens enviadas para esse canal. Teremos assim o *Control Receiver thread* (que está á

escuta no canal MC) e o *Data Backup Receiver thread* (que está á escuta no MDB). Para o *Chunk Backup Protocol* não é necessário interagir com o canal MDR.

Acrescidamente, cada um dos *threads* mencionados processa as mensagens recebidas, através do respectivo canal, de forma sequencial. Cada *thread* conclui o processamento de uma mensagem antes de iniciar o processamento da mensagem seguinte. Assim, cada um destes *threads* deve executar um *loop* infinito e em cada iteração deste trata de:

- receber uma mensagem no respectivo canal *multicast*;
- processar a mensagem recebida.

O envio de mensagens **PUTCHUNK** pelo *initiator peer* deve ser realizado por um *thread* diferente dos anteriores, i.e. o *thread multicaster*. Desta forma, o *thread multicaster* pode executar *Thread.sleep()* entre o *multicast* de mensagens **PUTCHUNK** consecutivas, implementando o intervalo de espera especificado no guião. Este thread pode ser criado em resposta a uma solicitação do *testClient*, para executar uma instância do *Chunk Backup Protocol*, e pode terminar quando a instância do protocolo acabar de se processar.

Se pretendem implementar a interface entre o *testClient* e o *peer* usando RMI, o *thread multicaster* pode ser o próprio *thread* criado pelo RMI *run-time* para lidar com a operação de *backup* invocada pelo *testClient*.

Caso contrário, ou seja, se pretendem empregar UDP ou TCP para essa interface, será necessário um *thread* adicional para receber os pedidos do *testClient*. Tal como os *Receiver threads*, este *thread* também pode processar os pedidos do *testClient* em série: ou seja, deve concluir o processamento de um pedido antes de iniciar o processamento do próximo (na verdade, se suportasse a execução simultânea de pedidos do *testClient*, violaríamos a suposição, acima expressa, de que existe no máximo uma instância do protocolo em execução a qualquer momento).

Notem que, no *initiator peer*, o *thread multicaster* e o *Control Receiver thread* podem precisar de partilhar informações (i.e. o número de mensagens **STORED** recebidas), para que o primeiro saiba como proceder. Compete-vos assim garantir que na comunicação entre os dois (por exemplo, no acesso aos objectos partilhados para a implementação dessa comunicação) não haja *race conditions* (problemas de concorrência).

Nota: nesta versão da implementação, os *peers* não iniciadores não precisam de processar as mensagens **STORED**. De qualquer das formas, a implementação de uma tal funcionalidade nesses *peers*, não seria assim tão complexa até porque já é algo que deverá ser feito ao nível do *peer* iniciador.

4. Um Thread por Canal M., Múltiplas Instâncias do Protocolo Simultâneas

Nesta versão da implementação considera-se que já deverá ocorrer mais do que uma execução simultânea do protocolo.

Utilizamos a mesma "arquitectura" de *threads* da versão anterior: existe assim um *Receiver Thread* por cada canal *multicast*, e cada um deles conclui o processamento de uma mensagem, antes de iniciar o processamento da próxima.

Embora possa haver mais do que um *thread multicaster* em execução, tanto no colectivo de *peers* como em cada peer individual (ou seja, pode haver vários peers a iniciar *backups* no sistema), cada um desses *threads* participa apenas na execução de uma instância do *Chunk Backup Protocol*.

De forma oposta, os *Receiver threads* podem precisar de processar mensagens relativas a uma instância do protocolo entre o processamento de mensagens relativas a outra instância do protocolo (para poderem lidar com vários pedidos simultâneos do colectivo de *peers*). Os *Receiver threads* devem, assim, empregar um objecto (estrutura de dados para armazenamento e partilha de informação), por instância de protocolo, para manter as informações relevantes para o processamento de mensagens pertencentes a essa instância de protocolo.
Esse objecto deverá ser construído aquando da transmissão/recepção da primeira mensagem da sua instância de protocolo, dependendo se o *peer* é o iniciador (transmissão) ou não (recepção) dessa instância do protocolo. Além disso, o objecto em questão deve ser mantido numa estrutura de dados, partilhada, de onde será recuperado sempre que isso seja necessário para processar cada uma das mensagens subsequentes da sua instância do protocolo de *Backup*. Desta forma, a primeira acção de um *Receiver thread* após receber uma mensagem (do seu canal *multicast*) deverá ser a recuperação (*retrieval*) do objecto (de armazenamento de dados) da instância do protocolo (*Backup*) à qual a mensagem pertence. Após isso poderá então usar as informações armazenadas nesse objeto para processar a mensagem.

Dica: empreguem uma instância da classe *java.util.concurrent.ConcurrentHashMap* para manter os objetos (armazenadores de informação a ser trocada entre *threads*) das diferentes instâncias de cada protocolo (i.e. um *ConcurrentHashMap* por protocolo).

5. Processando Diferentes Mensagens Recebidas Simultaneamente no mesmo Canal

Na versão anterior, as mensagens recebidas através de um determinado canal *multicast* eram tratadas em série.

Para suportar o processamento simultâneo de diferentes mensagens recebidas no mesmo canal, é necessário empregar mais do que um *thread* por canal. A solução típica passa pelo emprego de *Worker threads*: ou seja, empregar um *thread* diferente do *Receiver thread*, para processar as mensagens recebidas. Especificamente, o *Receiver thread* deve lançar um novo *thread* (*Worker* thread) para o processamento de cada mensagem recebida.

5.1. Uma Solução mais Escalável

O problema com a solução anterior é que a criação e o término de *threads* implica algum *overhead*, no qual se incorre uma vez por mensagem. Uma solução mais escalável é usar uma *pool* de *threads*, o que permite evitar a criação de um novo *thread* para processar cada mensagem.

Assim, esta iteração da implementação deverá empregar *thread pools*. Para isso, convém usar a classe *java.util.concurrent.ThreadPoolExecutor*.

6. Não recorrer a *Thread.sleep()*

O recurso a *Thread.sleep()* para implementação de *timeouts* pode levar à existência simultânea de um grande número de *threads* (em espera), cada um dos quais requer alguns recursos, limitando, portanto, a escalabilidade da solução.

Para evitar esta situação, poderão empregar a classe *java.util.concurrent.ScheduledThreadPoolExecutor*, a qual permite o agendamento de *timeouts*

7. Eliminar qualquer possibilidade de Bloqueio

Para uma escalabilidade máxima, deve ser eliminado o recurso a qualquer função bloqueante, nomeadamente, as chamadas de acesso ao sistema de ficheiros. Para isso poderão usar a classe *java.nio.channels.AsynchronousFileChannel*.

8. Conclusão

Este documento foca-se nos aspectos de concorrência do Projeto 1 e assume implicitamente que o ficheiro alvo do *backup* tem apenas um *chunk*. As sugestões antes apresentadas podem, de qualquer das formas, ser aplicadas ao *backup* de ficheiros com mais de um *chunk*, bem como à implementação dos outros subprotocolos.

Devem seguir uma abordagem incremental. No entanto, neste caso, vocês tem um "espaço de soluções" que é, pelo menos, bidimensional: numa das dimensões, está o nível de simultaneidade e escalabilidade pretendido para a vossa solução e, na outra dimensão, estão os subprotocolos (o número de *chunks* por arquivo adiciona, de certa forma, uma terceira dimensão).

A melhor abordagem será, provavelmente, uma do tipo "depth-first", na qual vocês primeiro implementam o *Chunk Backup Protocol* com o nível de simultaneidade que desejarem. Depois implementam os outros protocolos com o mesmo nível de simultaneidade. Devem, no entanto, gerir o vosso tempo: se demorarem muito para tratar da simultaneidade, depois talvez vos falte tempo para implementar os restantes protocolos.

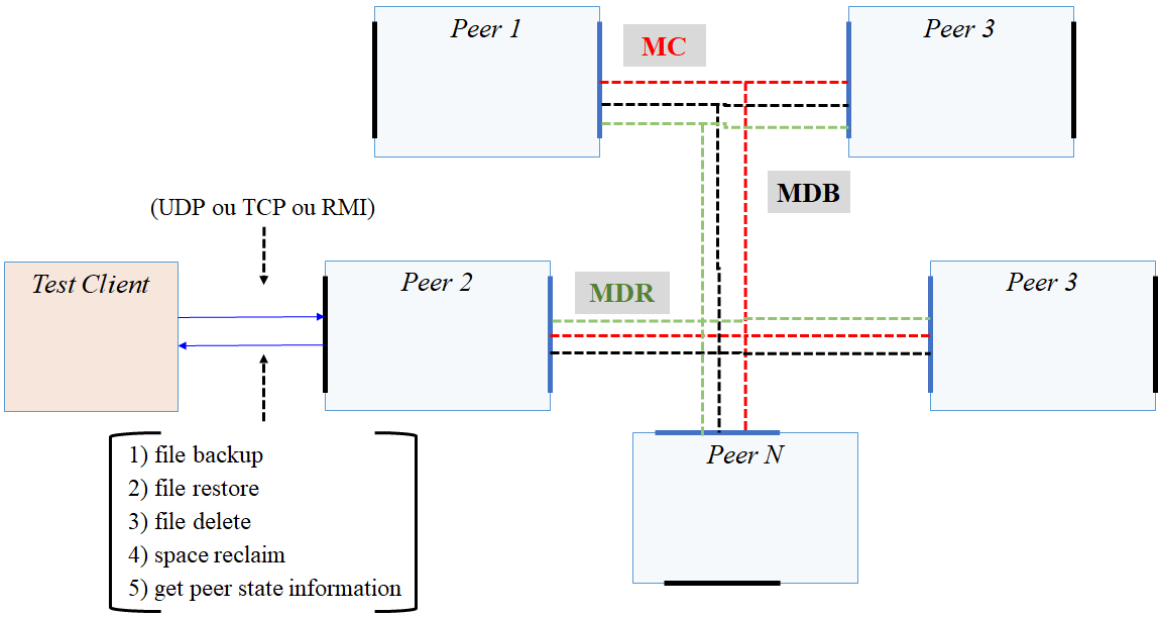
9. Leitura Adicional

Existem certamente muitos recursos bibliográficos relativos ao assunto que aqui abordamos. Os seguintes tutoriais parecem-me, no entanto, apresentar um bom equilíbrio entre teoria e prática e, portanto, podem ser úteis ao desenho e implementação do vosso primeiro projeto de SDIS (em Inglês):

- Java Concurrency/Multithreading Tutorial – A tutorial discussing concurrency issues in the context of Java. You may wish to take a look at the Concurrency Models chapter, which discusses several concurrency designs more abstractly than these notes.
- Java Concurrency Utilities – A tutorial on the *java.util.concurrent* package, including *ThreadPoolExecutor* and *ScheduledExecutorService*, which is the interface provided by the class *ScheduledThreadPoolExecutor*.
- Java NIO Tutorial – A good tutorial on Java NIO, including the *java.nio.channels.AsynchronousFileChannel*.

Notas sobre Trabalho 1

Já está disponível na página da cadeira o enunciado do primeiro projecto que vocês terão de desenvolver para avaliação.



Descrição sumária da estrutura e funcionamento do sistema a implementar:

- O trabalho a desenvolver constitui uma arquitectura P2P dedicada ao *backup* distribuido de ficheiros. Existirá assim um colectivo de *peers* que cooperará para armazenar de forma distribuida os ficheiros especificados, de acordo com um grau de replicação (número de cópias) também especificado. O sistema suportará também os serviços associados de *recovery*, *deletion* e *space reclaiming*;

- A arquitectura em questão compreende os dois seguintes tipos de componentes:
  - o *peer* – aplicação *p2p* que tanto recebe pedidos de outros *peers* como submete pedidos a outros *peers*. O sistema será assim composto por vários peers *idênticos* a correr simultaneamente e a interagir;
  - o *testClient* – aplicação que efectuará o interface entre o utilizador e um determinado *peer*. Apenas 1 *testClient* será empregue para intermediar a interacção entre o utilizador e um determinado peer;

- A comunicação entre os diferentes componentes proceder-se-á da seguinte forma:
  - o *testClient* comunica com o *peer* através (preferivelmente) de RMI;
  - os *peers* comunicam uns com os outros através de 3 canais multicast.

- Os canais multicast serão os seguintes:
  - MC – *Multicast Control Channel*. Canal empregue para a troca das mensagens de controlo do sistema. Todos os peers devem *estar à escuta* deste canal;
  - MDB – *Multicast Data Backup Channel*. Canal empregue nos procedimentos de *backup*;
  - MDR – *Multicast Data Recovery Channel*. Canal empregue nos procedimentos de *recovery*;

- Sobre a arquitectura descrita deverão ser suportados 4 serviços (protocolos) relacionados (oferecidos pelo *testClient* ao utilizador):
  - *Backup* – para replicar um ficheiro com um grau de replicação especificado;

  - Recovery – para recuperar um ficheiro que foi previamente replicado;
  - *Delete* – para eliminar do sistema um ficheiro antes replicado;
  - *Space Reclaiming* – para libertar espaço (disco) de um *peer* garantindo a preservação do *replication degree* dos ficheiros (*chunks*, para ser mais correcto) por ele armazenado;

- A operação do sistema será a seguinte:
  - o utilizador solicita um dos serviços suportados ao *testClient*;
  - o *testClient* constrói e envia ao *peer* o pedido correspondente;
  - o *peer* que recebe o pedido (designado como *initiator peer*) interage com os restantes *peers* (através dos canais *multicast* adequados) para levar a cabo os necessários procedimentos para assegurar o serviço solicitado (*backup*, *recovery*, *delete* ou *space reclaim*). Aos procedimentos desencadeados para suportar um determinado serviço (ou protocolo) chama-se (no guião) um sub-protocolo;
  - notem que o suporte de um determiando serviço pode despoletar um vasto conjunto de interacções envolvendo outros *peers* que não o próprio *initiator peer*;

- É de notar que (no contexto das operações/interacções que se desenvolvem para dar suporte aos serviços oferecidos pelo sistema) os *peers* não operarão sobre os ficheiros inteiros, mas sobre fragmentos destes, designados de *chunks*.
  Assim, aquando do *backup*, o ficheiro inteiro será fornecido ao *testClient* (pelo utilizador), que o fará chegar ao seu *peer*. Nesse ponto cada *peer* deverá fragmentar cada ficheiro no numero necessário de *chunks* para que cada um destes possa ser transportado no *payload* de um datagrama UDP (para poder circular nos canais de comunicação entre peers).
  Depois o *peer* deverá desenvolver o mesmo procedimento para cada *chunk* (*backup*). Estes procedimentos deverão ser executados concorrentemente, usando múltiplos *threads*.

- Da mesma forma, a recuperação (*recovery*) de um ficheiro implicará a recuperação individual de dada um dos *chunks* em que este foi dividido aquando do *backup*. O *initiator peer* tratará de obter (pedir aos outros *peers*) todos os *chunks* de um ficheiro, reconstituirá o ficheiro e só depois o devolverá ao *testClient* (e, portanto, ao utilizador);

- Em face do que é expresso acima fica claro que o *peer* deverá estar à escuta de pedidos vindos de dois lados: do *testClient* (através de uma ligação RMI) e do colectivo de *peers* (através dos 3 canais multicast). Um peer deve assim apresentar dois interfaces:
  - o interface para o *testClient* – serviço que permita ao *testClient* a submissão de pedidos de *backup*, *recovery*, *deletion*, ou *space reclaim* (preferencialmente usando Java RMI (vale 5%), mas pode no entanto ser implementado de outra forma, p.ex. usando TCP ou UDP);
  - o interface para os restantes peers – aqui referimo-nos aos mecanisos do *peer* que estão à escuta dos pedidos provenientes dos 3 canais multicast, portanto, à escuta de pedidos dos outros peers;

A acção dentro de um *peer* pode portanto ser desencadeada por um pedido vindo do *testClient* (utilizador) ou de um outro *peer*.

Notas gerais:

- Como não é possível testar os restantes protocolos/serviços sem implementar o protocolo de *Backup*, o recomendável é que comecem por implementar este protocolo. Para além disso, embora se pretenda uma implementação concorrente baseada em múltiplos *threads*, inicialmente podem fazer uma implementação mais simples, capaz de fazer o *backup* de um ficheiro com um *chunk* apenas;
- A ordem gobal mais adequada para implementarem os sub-protocolos é: *backup*, *delete*, *restore* e *reclaim*;

Notas sobre Trabalho 1

- Devem fazer uma implementação incremental. Ou seja implementar cada funcionalidade necessária isoladamente (p.ex. enviar/receber mensagens em multicast, calcular o identificador do ficheiro para *backup*, partir o ficheiro em *chunks*, etc.) e ir integrando essas funcionalidades gradualmente;
- Há um conjunto de possíveis *enhancements*, referidos ao longo do enunciado, que vocês podem implementar. Devem é implementar essas melhorias depois de completarem as versões base de todos os protocolos.
- A demonstração do vosso projecto requererá que seja possível executar os vossos *peers*, assim como o *testClient*, a partir da linha de comandos. Recomendamos assim que vocês criem um script para agilizar esse processo;
- É também necessário, para efeitos de teste, que a vossa implementação permita executar múltiplos *peers* no mesmo computador;
- Para além disso, a demonstração será realizada em PCs da sala de aula, assim para não serem penalizados por problemas súbitos que surjam durante a demonstração ensaiem o *setup* do vosso sistema antes da demonstração;
- Notem que a montagem do vosso sistema para demonstração valerá 5% da nota do projecto;
- Um dos testes a que os vossos *peers* serão sujeitos é um teste de interoperabilidade com os *peers* de outros grupos. Assim, podem e devem realizar esses testes entre as vossas implementações. No entanto, para o fazerem podem partilhar apenas ficheiros *.class* (pré-binários java) com outros grupos e não ficheiros *.java* (código fonte).
- Relativamente ao *replication degree* é de notar que o valor especificado (aquando do *backup*) é apenas o valor desejado. Um *chunk* pode acabar por ser replicado com um grau inferior, se p.ex. não houver um número suficiente de *peers*, ou superior.