Intergiciels et Communication Distribuée

1. Introduction aux Intergiciels (Middleware)

Les intergiciels sont des couches logicielles permettant la communication et l'échange de données entre applications distribuées. Ils masquent la complexité du réseau et facilitent l'interopérabilité entre différents systèmes.

Types d'intergiciels :
- Intergiciels de communication (Sockets, RMI, JMS, etc.).
- Intergiciels transactionnels (CORBA, JTA).
- Intergiciels orientés services (SOAP, REST, gRPC).
- Intergiciels de persistance (JDBC, Hibernate).

2. Sockets (Communication Bas Niveau)

Les sockets permettent la communication entre applications via un réseau en utilisant les protocoles TCP/IP ou UDP.

Différences TCP vs UDP :

TCP (Transmission Control Protocol) :
- Orienté connexion
- Fiabilité garantie
- Plus lent mais sûr (Web, FTP, SSH...)

UDP (User Datagram Protocol) :
- Non connecté
- Pas de garantie de livraison
- Rapide, utilisé pour le streaming, DNS, VoIP

3. RMI (Remote Method Invocation)

RMI permet d'exécuter des méthodes à distance sur des objets situés sur un autre serveur Java.
Composants principaux :
- Serveur RMI : Héberge les objets distants.
- Client RMI : Appelle les méthodes distantes.
- RMI Registry : Annuaire d'enregistrement des objets distants.

Intergiciels et Communication Distribuée

4. JMS (Java Message Service)

JMS est une API Java permettant la communication asynchrone via des messages.
Deux modèles de communication :
- Point à Point (P2P) : Un seul consommateur reçoit le message.
- Publication/Abonnement (Pub/Sub) : Plusieurs abonnés reçoivent les messages.

5. Space Linda (Modèle de Coordination)

Linda est un modèle de programmation concurrente basé sur un espace partagé (Tuple Space) pour la communication entre processus.

Opérations principales :
- `out(tuple)`: Insère un tuple.
- `in(template)`: Récupère et supprime un tuple correspondant.
- `rd(template)`: Lit un tuple sans le supprimer.

6. Comparaison des Technologies

Tableau récapitulatif des intergiciels :

Technologie	Type	Communication	Utilisation principale
Sockets	Bas niveau	Synchrone	Communication réseau brute
RMI	Objets distants	Synchrone	Invocation de méthodes distantes
JMS	Message-Oriented	Asynchrone	Échange de messages en entreprise
Linda	Coordination	Asynchrone	Espace partagé pour coordination

Introduction to Sockets in C

Generated with LaTeX

March 20, 2025

1 Introduction

Sockets provide a way for programs to **communicate over a network**. They support two main protocols:

- **TCP (Transmission Control Protocol)**: Connection-oriented, reliable, ordered data transmission.
- **UDP (User Datagram Protocol)**: Connectionless, fast but unreliable.

2 Basic TCP Server Implementation

The following C program creates a **basic TCP server** that listens for connections and reads data from a client.

Listing 1: Basic TCP Server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);
    char buffer[1024] = {0};

    // Create socket file descriptor
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
```

```c
    if (server_fd == -1) {
        perror("Socket␣failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind socket to port
    bind(server_fd, (struct sockaddr *)&address, sizeof(
        address));

    // Listen for connections
    listen(server_fd, 3);
    printf("Listening␣on␣port␣%d...\n", PORT);

    // Accept connection
    new_socket = accept(server_fd, (struct sockaddr *)&
        address, &addrlen);
    read(new_socket, buffer, 1024);
    printf("Message␣received:␣%s\n", buffer);

    close(new_socket);
    close(server_fd);
    return 0;
}
```

3 Basic TCP Client Implementation

The following C program creates a **basic TCP client** that connects to a server and sends a message.

Listing 2: Basic TCP Client

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char *message = "Hello,␣Server!";

```

```c
    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);

    // Connect to server
    connect(sock, (struct sockaddr *)&serv_addr, sizeof(
        serv_addr));

    // Send message
    send(sock, message, strlen(message), 0);
    printf("Message sent\n");

    close(sock);
    return 0;
}
```

4 Handling Multiple Clients

A server can handle multiple clients using **fork()** or **threads**. Below is a **multi-client server** using **fork()**.

Listing 3: Handling Multiple Clients with fork()

```c
void handle_client(int client_socket) {
    char buffer[1024] = {0};
    read(client_socket, buffer, 1024);
    printf("Client: %s\n", buffer);
    close(client_socket);
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    bind(server_fd, (struct sockaddr *)&address, sizeof(
        address));
    listen(server_fd, 5);

    while (1) {
        new_socket = accept(server_fd, (struct sockaddr *)&
            address, &addrlen);
        if (fork() == 0) { // Child process
            handle_client(new_socket);
```

```
            exit(0);
        }
    }

    close(server_fd);
    return 0;
}
```

5 TCP vs. UDP Connections

The key differences between **TCP** and **UDP** connections:

- **TCP** ensures **reliable** and **ordered** data delivery.
- **UDP** is **faster**, but packets may be lost or arrive **out of order**.

UDP Server Example:

Listing 4: Basic UDP Server

```
int sockfd;
struct sockaddr_in servaddr, cliaddr;
char buffer[1024];

// Create UDP socket
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

// Receive data
recvfrom(sockfd, buffer, 1024, 0, (struct sockaddr *)&cliaddr
    , &len);
printf("Received: %s\n", buffer);
```

UDP Client Example:

Listing 5: Basic UDP Client

```
int sockfd;
struct sockaddr_in servaddr;
char *message = "Hello UDP Server";

// Create UDP socket
sockfd = socket(AF_INET, SOCK_DGRAM, 0);

// Send message
sendto(sockfd, message, strlen(message), 0, (struct sockaddr
    *)&servaddr, sizeof(servaddr));
```

6 Read and Write Operations

Sockets support **read()** and **write()** for TCP and **sendto()**/**recvfrom()** for UDP.

TCP Read and Write:

Listing 6: TCP Read/Write

```
char buffer[1024];
int bytes_read = read(socket_fd, buffer, sizeof(buffer));
write(socket_fd, "Message Received", 16);
```

UDP Send and Receive:

Listing 7: UDP Send/Receive

```
sendto(socket_fd, message, strlen(message), 0, (struct
    sockaddr *)&server_addr, sizeof(server_addr));
recvfrom(socket_fd, buffer, sizeof(buffer), 0, (struct
    sockaddr *)&client_addr, &len);
```

7 Conclusion

This document introduced:

- **Basic TCP Server and Client**
- **Handling Multiple Clients**
- **TCP vs. UDP Differences**
- **Read and Write Operations in Both Protocols**

Sockets provide **low-level network communication** and are used in **web servers, gaming, real-time messaging, and networking applications**.

Java RMI Example: Remote Method Invocation

Generated with LaTeX

March 20, 2025

1 Introduction

This document provides an example of a **Java RMI (Remote Method Invocation)** system, implementing a distributed contact book (‘Carnet‘). It includes:

- A **server** that registers the ‘CarnetImpl‘ object.
- A **serializable individual** (‘IndividuSerializable‘).
- A **remote individual** (‘IndividuRemote‘).
- The **interface** defining remote methods.
- The **implementation** of the contact book (‘CarnetImpl‘).

2 Server Code

2.1 Creating the RMI Server

The following code creates the **RMI registry** and registers an instance of ‘CarnetImpl‘:

Listing 1: ServeurCarnet.java

```java
package carnet;

import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class ServeurCarnet {
```

```java
    public static void main (String args[]) throws Exception {
        try {
            LocateRegistry.createRegistry(1099);
        } catch (java.rmi.server.ExportException e) {
            System.out.println("A registry is already running, proceeding...");
        }

        CarnetImpl carnet = new CarnetImpl();
        Naming.rebind("rmi://localhost:1099/MonCarnet", carnet);

        System.out.println("Le systeme est pret.");
    }
}
```

3 Serializable Individual

The ‘IndividuSerializable‘ class implements ‘Serializable‘, allowing it to be transferred over the network.

Listing 2: IndividuSerializable.java

```java
import java.io.Serializable;

public class IndividuSerializable implements Individu, Serializable {
    private static final long serialVersionUID = 1L;
}
```

4 Remote Individual

The ‘IndividuRemote‘ class extends ‘UnicastRemoteObject‘ to be accessible remotely.

Listing 3: IndividuRemote.java

```java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class IndividuRemote extends UnicastRemoteObject implements Individu {
    public IndividuRemote() throws RemoteException {}
}
```

5 Carnet Interface

The ‘Carnet‘ interface defines the remote methods that can be invoked by clients.

Listing 4: Carnet.java

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Carnet extends Remote {
    void inserer (IndividuSerializable x) throws RemoteException;
    IndividuRemote chercher (String nom) throws RemoteException, IndividuInexistant;
    IndividuRemote get (int n) throws RemoteException, IndividuInexistant;
    IndividuRemote[] getAll() throws RemoteException;
    void addCallbackOnCreation(CallbackOnCreation cb) throws RemoteException;
    void removeCallbackOnCreation(CallbackOnCreation cb) throws RemoteException;
}
```

6 Carnet Implementation

The ‘CarnetImpl‘ class implements the ‘Carnet‘ interface.

Listing 5: CarnetImpl.java

```java
import java.rmi.RemoteException;
import java.util.ArrayList;
import java.util.List;

public class CarnetImpl implements Carnet {
    private List<IndividuRemote> contenu = new ArrayList<>();
    private List<CallbackOnCreation> callbacks = new ArrayList<>();

    @Override
    public void inserer(IndividuSerializable x) throws RemoteException {
        IndividuRemote i = new IndividuRemote(x.nom(), x.age());
        contenu.add(i);
        for (CallbackOnCreation cb : callbacks) {
            cb.eventCreated(x);
        }
```

```
    }

    @Override
    public IndividuRemote chercher(String nom) throws
        RemoteException, IndividuInexistant {
        for (IndividuRemote i : contenu) {
            if (i.nom().equals(nom)) {
                return i;
            }
        }
        throw new IndividuInexistant("Individu inexistant");
    }
}
```

7 Conclusion

This document provides a complete example of an **RMI-based distributed contact book**. The system allows remote insertion, searching, and listing of individuals in the carnet.

Java JMS Example: Message-Oriented Middleware

Generated with LaTeX

March 20, 2025

1 Introduction

This document provides an example of **Java Message Service (JMS)** implementation using **JORAM**. The system includes:

- **Queue vs. Topic Explanation**: Key differences between the two messaging models.
- **Destination Creation**: Setting up the JMS server with topics or queues.
- **Message Sending and Receiving**: Using producers and consumers.
- **Chat Application (IRC-like)**: A messaging system with public and private messaging, including a "Who" command.

2 Queue vs. Topic: Key Differences

JMS supports two messaging models:

- **Queue (Point-to-Point, P2P)**: Each message is consumed by a **single receiver**. Messages are stored in a queue and delivered once.
- **Topic (Publish-Subscribe, Pub/Sub)**: Messages are broadcast to **multiple subscribers**. Each subscriber gets a copy of the message.

Listing 1: Creating a Queue or Topic

```
Destination queue = Queue.create(0); // Point-to-Point (One consumer)
Destination topic = Topic.create(0); // Publish-Subscribe (Multiple consumers)
```

3 Destination Creation

The following code sets up a JMS destination and binds it in **JNDI**.

Listing 2: CreateDestination.java

```java
package irc;

import org.objectweb.joram.client.jms.admin.*;
import org.objectweb.joram.client.jms.*;
import org.objectweb.joram.client.jms.tcp.*;

public class CreateDestination {
  public static void main(String args[]) throws Exception {
    System.out.println("CreateDestination administration phase... ");

    // Connecting to JORAM server:
    AdminModule.connect("root", "root", 60);

    // Creating the JMS administered objects:
    javax.jms.ConnectionFactory connFactory =
      TcpConnectionFactory.create("localhost", 16010);

    Destination destination = Topic.create(0);
    // Destination destination = Queue.create(0); // For queues

    // Creating an access for user anonymous:
    User.create("anonymous", "anonymous", 0);

    // Setting free access to the destination:
    destination.setFreeReading();
    destination.setFreeWriting();

    // Binding objects in JNDI:
    javax.naming.Context jndiCtx = new javax.naming.InitialContext();
    jndiCtx.bind("ConnFactory", connFactory);
    jndiCtx.bind("MonTopic", destination);
    jndiCtx.close();

    AdminModule.disconnect();
    System.out.println("Admin closed.");
  }
}
```

4 Sending and Receiving Messages

This code demonstrates how a **producer** sends a message and a **consumer** receives it asynchronously.

Listing 3: HelloWorld.java

```java
package irc;

import javax.jms.*;
import javax.naming.*;

public class HelloWorld {
    public static void main(String argv[]) {
        try {
            InitialContext ic = new InitialContext();

            ConnectionFactory connectionFactory = (ConnectionFactory) ic.lookup("ConnFactory");
            Destination destination = (Destination) ic.lookup("MonTopic");

            System.out.println("Bound to ConnFactory and MonTopic");

            Connection connection = connectionFactory.createConnection();
            connection.start();

            System.out.println("Created connection");

            System.out.println("Creating sessions: not transacted, auto ack");
            Session sessionP = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
            Session sessionS = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

            MessageProducer producer = sessionP.createProducer(destination);
            MessageConsumer consumer = sessionS.createConsumer(destination);

            consumer.setMessageListener(new MessageListener() {
                public void onMessage(Message msg) {
                    try {
                        TextMessage textmsg = (TextMessage) msg;
```

```java
                        System.out.println("I have received : " + textmsg.getText());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            });

            System.out.println("Ready");

            TextMessage textmsg = sessionP.createTextMessage();
            textmsg.setText("Hello World !!!");
            producer.send(textmsg);

        } catch (Exception ex) {
            ex.printStackTrace();
            return;
        }
    }
}
```

5 Chat System with "Who" Command

The following code implements an **IRC-like chat system** using JMS, allowing users to:

- Send public messages
- Send private messages
- List all active users with the "Who" command

Listing 4: Who Command in Irc.java

```java
private class WhoListener implements ActionListener {
    public void actionPerformed (ActionEvent ae) {
        System.out.println("who button pressed");
        try {
            print("Connected users:");
            for (String user : activeUsers) {
                print("- " + user);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

6 Conclusion

This document demonstrates how **Java Message Service (JMS)** can be used for:

- Setting up a messaging system with **JORAM**.
- Understanding the difference between **Queue (P2P) and Topic (Pub-/Sub)**.
- **Sending and receiving messages** asynchronously.
- Implementing a **real-time chat application** with public and private messaging, and the "Who" command.

JMS is an essential tool for enterprise applications, providing **scalability and decoupled communication** between services.

Introduction to Linda Spaces with Java

Generated with LaTeX

March 20, 2025

1 Introduction

Linda Spaces provide a **coordination model** for distributed systems, allowing processes to exchange **tuples** through a **shared TupleSpace**. The model is **asynchronous**, **decentralized**, and supports **synchronization between processes**.

Key operations in Linda Spaces:

- **out(tuple)** - Inserts a tuple into the space.
- **in(template)** - Removes a matching tuple (blocking).
- **rd(template)** - Reads a tuple without removing it (blocking).
- **inp(template)** - Non-blocking version of **in()**.
- **rdp(template)** - Non-blocking version of **rd()**.

2 Connecting to a TupleSpace

To connect to a **TupleSpace**, the client must specify the server's hostname and port.

Listing 1: Connecting to TupleSpace

```
import com.ibm.tspaces.*;

public class TupleSpaceExample {
    public static void main(String[] args) {
        try {
            TupleSpace space = new TupleSpace("Whiteboard", "
                localhost", 6000);
            System.out.println("Connected to TupleSpace!");
```

```
        } catch (TupleSpaceException e) {
            e.printStackTrace();
        }
    }
}
```

3 Writing Tuples to the Space

A tuple is a **structured collection of objects** (e.g., strings, numbers, booleans). Writing a tuple makes it available for all processes.

Listing 2: Writing a Tuple

```
space.write(new Tuple("Whiteboard", "DRAW", "Line1"));
```

4 Reading Tuples (Blocking and Non-Blocking)

Processes can read tuples without removing them using **rd()** or **rdp()**.

Listing 3: Reading a Tuple

```
Tuple template = new Tuple("Whiteboard", "DRAW", String.class
    );
Tuple result = space.rd(template); // Blocking read
System.out.println("Tuple found: " + result.getField(2));
```

For a **non-blocking read**, use **rdp()**:

Listing 4: Non-blocking Read

```
Tuple result = space.rdp(template);
if (result != null) {
    System.out.println("Tuple found: " + result.getField(2));
} else {
    System.out.println("No matching tuple found.");
}
```

5 Removing Tuples

The **in()** operation removes a matching tuple from the TupleSpace.

Listing 5: Removing a Tuple

```
Tuple result = space.in(template);
System.out.println("Removed tuple: " + result.getField(2));
```

For a **non-blocking removal**, use **inp()**:

Listing 6: Non-blocking Removal

```
Tuple result = space.inp(template);
if (result != null) {
    System.out.println("Removed␣tuple:␣" + result.getField(2)
        );
} else {
    System.out.println("No␣matching␣tuple␣found.");
}
```

6 Synchronization and Coordination

To ensure exclusive access, a process can use a **LOCK tuple**.

Listing 7: Lock Mechanism

```
Tuple lock = new Tuple("Whiteboard", "LOCK");
space.write(lock);  // Lock acquired

// Perform operations...

space.in(lock); // Release the lock
```

7 Listening for Events

Processes can listen for tuple updates using **callbacks**.

Listing 8: Listening for Events

```
space.eventRegister(TupleSpace.WRITE, new Tuple("Whiteboard",
    "DRAW", String.class), new EventDraw());
```

The callback function:

Listing 9: Event Callback Function

```
private class EventDraw implements com.ibm.tspaces.Callback {
    public boolean call(String eventName, String tsName, int
        sequenceNumber, SuperTuple tuple, boolean isException)
        {
        if (isException) return true;
        System.out.println("New␣drawing␣command␣received:␣" +
            tuple.getField(2));
        return false;
    }
}
```

8 Conclusion

Linda Spaces provide a powerful **coordination model** for distributed applications. This document demonstrated:

- **Writing and reading tuples**
- **Blocking and non-blocking operations**
- **Tuple removal and synchronization**
- **Listening for real-time events**

Using **TupleSpace**, applications can easily implement **distributed synchronization, blackboard architectures, and parallel computing**.