

---

# 2II45 - Architecture of Distributed Systems

## Assignment 2

---

Vincent Nuttin - v.m.nuttin@student.tue.nl  
Erasmus Student 2010-2011  
UCLouvain.be - Belgium

October 11, 2010

### Contents

<b>1</b>	<b>Basics</b>	<b>1</b>
<b>2</b>	<b>First choice : Peer-to-peer architecture</b>	<b>3</b>
2.1	How does it work ? . . . . .	3
2.2	Extra-functional aspects - Scalability and performance . . . . .	5
<b>3</b>	<b>Second choice : Client-Server architecture</b>	<b>6</b>
3.1	How does it work ? . . . . .	6
3.2	Extra-functional aspects - Scalability and performance . . . . .	8

# 1 Basics

A chat application can be built following different approaches. I will try to explain in more details two of them : a peer-to-peer approach and a client-server one.

Even though different models are possible, there remains some functionalities in common.

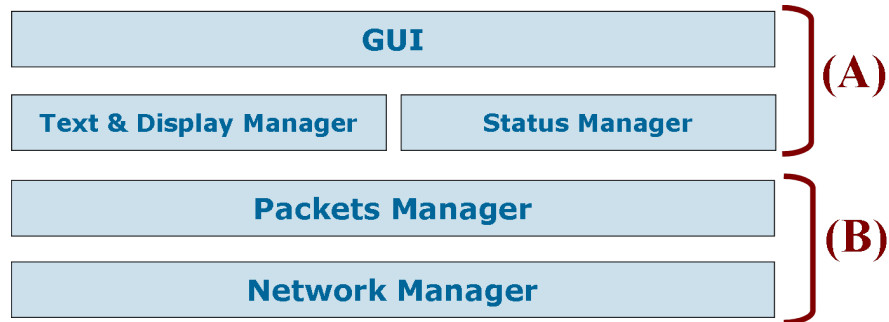


Figure 1: Some kind of development view. Part A remains the same in all architectural styles. Part B must be different in case of a peer-to-peer or a client-server architecture.

Notes on this schema :

- **Network Manager** (NM) : Responsible for listening to the network,
- **Packets Manager** (PM) : Responsible for classifying packets between system and normal chat messages,
- **Text and Display Manager** (TDM) : Responsible for normal chat messages received from the PM,
- **Status Manager** (SM) : Responsible for status, login and friend messages.

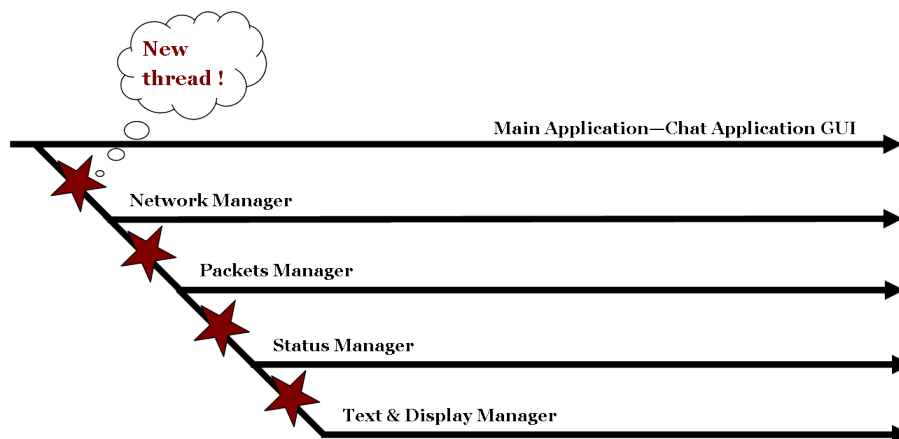


Figure 2: Process view

The most important thing is to run all these modules in separate threads because it is really important to deal with and display messages when the network manager is sending another one, etc.

You can see, on picture 3, the global shape of the two already mentioned architectures. I will try to explain a little bit more how these two architectures can be developed and how we can deal with some aspects of the process (e.g. login, discovery, friend relationships, etc.)

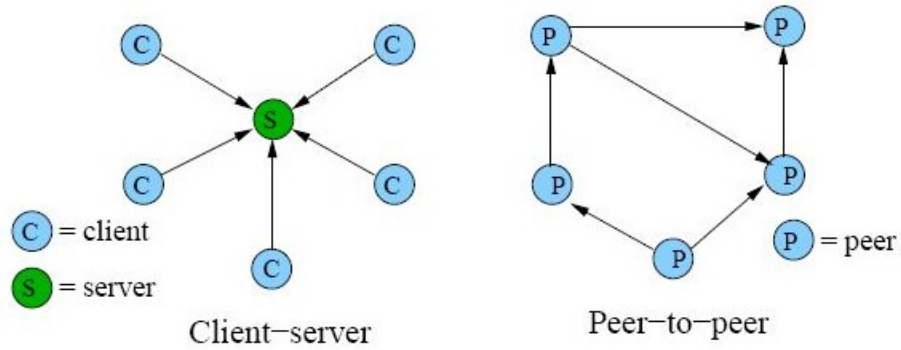


Figure 3: Picture from an article about telephony using SIP [2]

## 2 First choice : Peer-to-peer architecture

The first possibility to develop a chat application is the use of a peer-to-peer architecture.



Figure 4: A peer-to-peer system of nodes - Wikipedia.org

### 2.1 How does it work ?

Before all, I will assume that I work with a pure peer-to-peer network; I mean the system is composed by peers and links between them only. I don't want to use the functionalities of super-peers because I want to keep this system as basic and simple as possible.

Every node must maintain two separated lists. A first one containing names of friends (`List<Friend1, Friend2, Friend3, ...>`); and another list containing all online peers that we know (`List<[User1, IP, Port], [User2, IP, Port], [User3, IP, Port], ...>`). The friends-list is modified when we add or remove a friend. This process will be explained in more details in section 2.1.2. The online-list is managed by the status manager (see figure 1). In a first time, we add peers into this list during the login phase. But, after that, we can use an eventually perfect failure detector (EPFD) [1] to remove peers when they leave or they crash.

### 2.1.1 Login and Who is on-line ?

Let's take an example (with five peers already connected on the network and I'm the sixth one, arriving on the network) and let's make some assumptions :

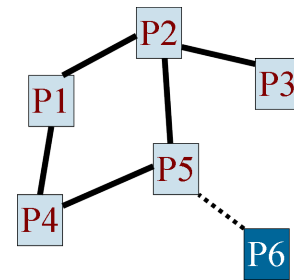


Figure 5: Example

- Let's assume that the discovery process is made easier thanks to a public place where connected peers can publish their "point-of-entrance" in the network. Whatever happens, we need to know at least one entry point.
- Let's assume that there exists a (well managed) DHT (*Distributed Hash Table*) containing all credentials of registered people of this chat service.

Peer 6 chooses Peer 5 as entry point. Peer 6 will send a join message to Peer 5 :

JOIN(Peer6, Password, IP, Port)

Figure 6 shows the behaviour of every node when a JOIN message arrives.

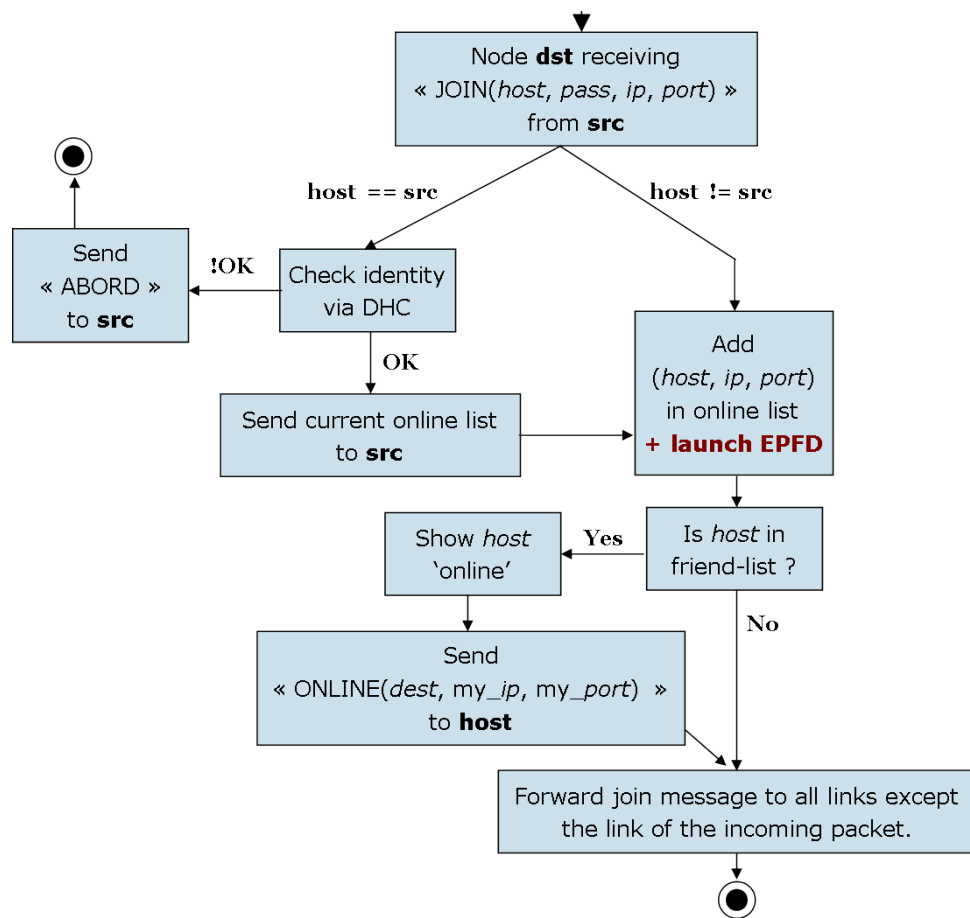


Figure 6: Behaviour model - In addition to that, we can imagine a way to avoid JOIN messages to loop forever in the network with a simple 'id' feature or something.

Thanks to this login process, we can retrieve the list of already connected people and launch our EPFD on every host to detect when they leave or they crash. We are now ready to begin chat sessions with connected friends.

### 2.1.2 Ask for friend relation

When the application is running and the user fully logged-in, we can simply search in the online list to find new friends. Then, we can directly send a ASK-FRIEND message to the IP address and the port of the new friend. This kind of message is treated by the packets manager (see figure 1). If the new friend agrees with this new relationship, he replies with a FRIEND-OK message to the original peer. If the new friend don't want to begin a relationship with the peer, he simply never replies to the message<sup>1</sup>. This process is very simple thanks to the fact that we are in a peer-to-peer network, where every node can act as a server, client or both. The communication between peers can be done directly thanks to the information contained in the online-list.

### 2.1.3 Chatting

In the same way than the friend messages, peers can simply exchange normal chat messages with each other thanks to the fact that they have all the information to join other peers.

We can easily understand here the importance of the packets manager (see figure 1). Actually, when a packet arrives in a node, the packet manager can simply read the header of this packet to know if it is a system notification (friends, status, etc.) or a real chat message (containing text to display).

## 2.2 Extra-functional aspects - Scalability and performance

In one hand, this choice (peer-to-peer architecture) is really good for scalability because, at no point of the architecture, the number of node is important. I think that we don't have to change anything if we use this chat application with 5 people or 50 people. The only little thing in relation with the number of node is the list of online people. We can imagine a better data structure to store this information (local hash-map, etc.) but with the high volume of space we have in nowadays computers, I think that even a list of 1000000-entries is not a problem to store. Once again, if we need to search in this list very often, we can imagine a better data structure than a list.

In the other hand, this choice is not the best one for the login performance. Actually, we have to wait until our JOIN message reaches the most far node to be fully connected to the network. I mean, if we have a global network with a high latency and a shortest path from us to the most far node of about 50 hops, we must wait 50 times the mean latency of the network to be sure that everybody on the network is aware of our presence.

---

<sup>1</sup>Here, we can imagine a simple process to deal with the loss of packets. The source can have a timeout feature to automatically re-send the ASK-FRIEND message but maximum twice or something.

### 3 Second choice : Client-Server architecture

The second possibility to develop a chat application is the use of a client-server architecture.



Figure 7: A centralized server-based system of nodes. - Wikipedia.org

#### 3.1 How does it work ?

My opinion is that this second architecture is simpler to achieve. Globally, this solution is more centralised and then, easier to understand.

Figure 8 shows the main modules of a client and the server (let's assume that the server is unique on the network).

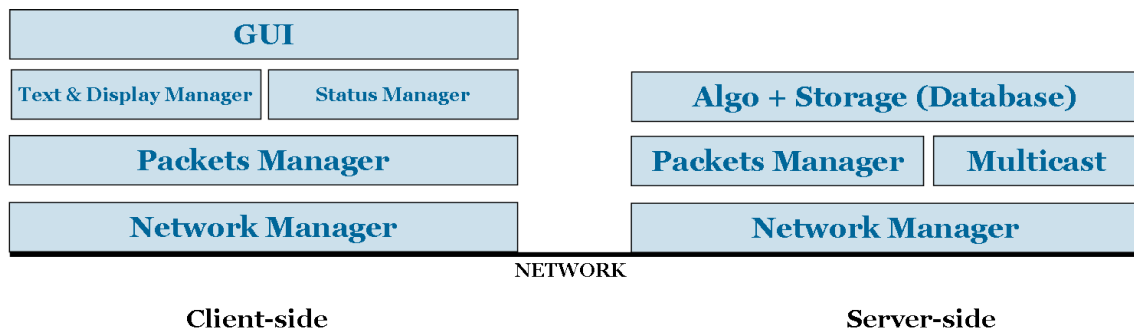


Figure 8: Main modules

An important thing to notice is that the server might use a reliable multicast to contact every node connected to him. This kind of behaviour will happen every time a (new) node is connecting or disconnecting.

### 3.1.1 Login and Who is on-line ?

The login phase consists in simple messages send to the (only one) server on the network. Everything is centralized so the server has just to check in a local database if the credentials of the user are correct. If they are, first the server replies with a JOIN\_OK message to the client. The message contains the list of connected friends. And secondly, the server sends a notification to every node which have the new node in its friend list. Here, we can understand the usage of the multicast module.

A global scenario :

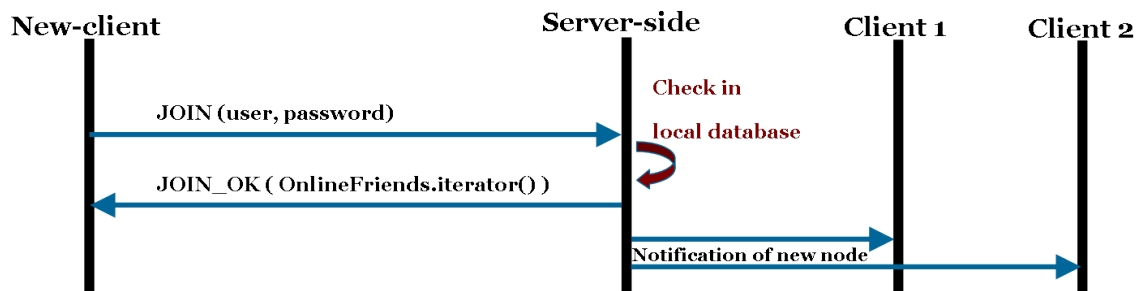


Figure 9: The login phase

and more specifically for the server :

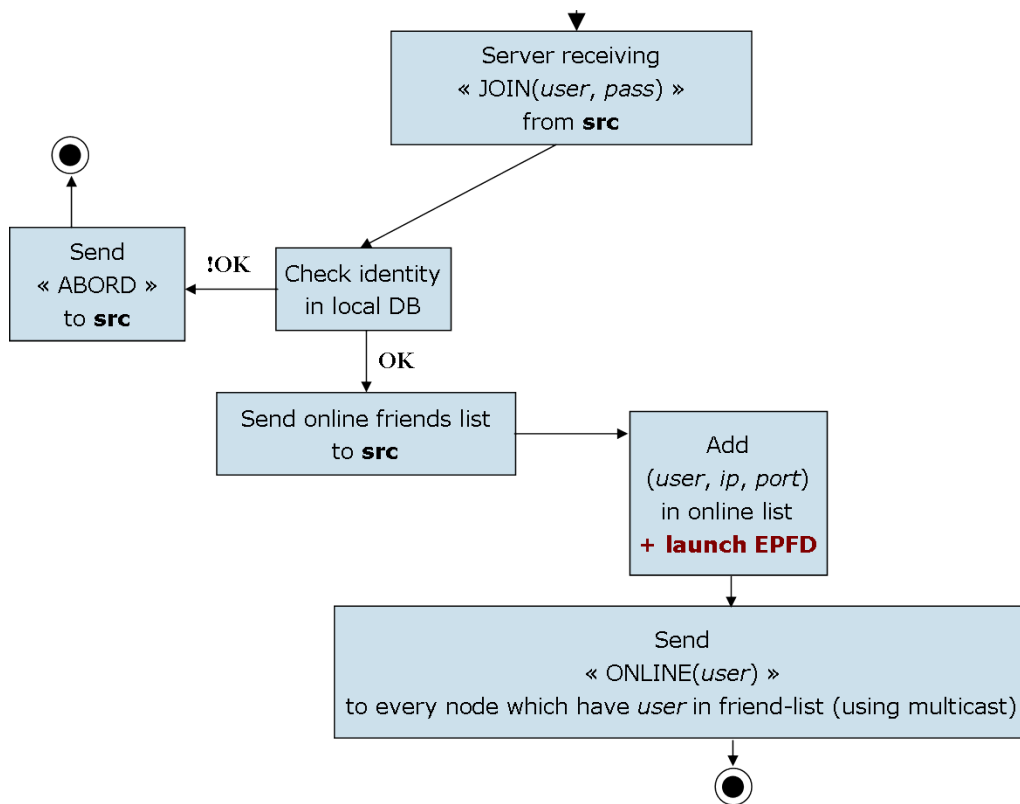


Figure 10: Behaviour model

### 3.1.2 Ask for friend relation

Once again, this process is simpler than in the peer-to-peer architecture :

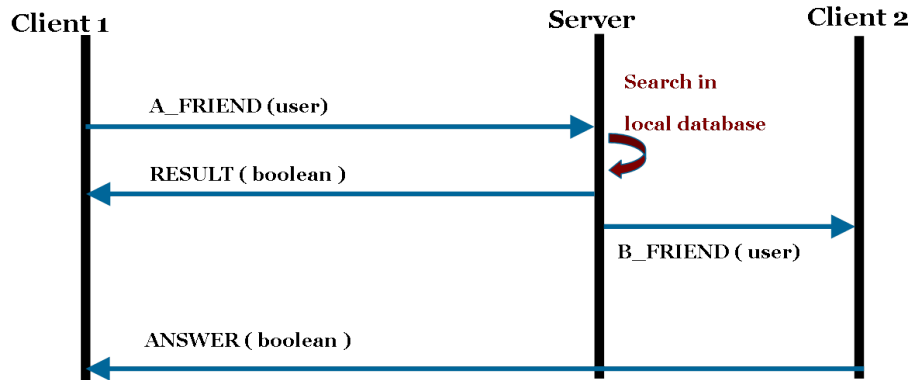


Figure 11: Ask for a friend relationship - Note that the server replies with a `RESULT` message to indicate to the client if the searched user exists or not. The reply to the invitation of relationship is a `ANSWER` message.

### 3.1.3 Chatting

When there is a friend who is online, you can begin a chat session with him. The process is really simple again. You always send your message to the server with its destination and the content. Then, the server just forward the content to the destination. As in the peer-to-peer architecture, the packets manager has a very important role to classify system notifications and real chat messages.

## 3.2 Extra-functional aspects - Scalability and performance

An application like that is not really effective. The major problem is the fact that everything is centralized and we cannot imagine a number of node growing up without a failure of the central server. In this case, the entire chat application is down. Also, we should add other servers to be able to reply to all the queries in a respectable delay.

Nevertheless, this architecture provide much better performance for the login phase thanks to the small number of hops needed to join all peers. This is the advantage of a centralized server. But, to conclude, I think that the drawbacks are bigger than the advantages.

## References

- [1] Seif Haridi and Peter Van Roy. Failure detectors. Royal Institute of Technology and Universite catholique de Louvain.
- [2] Kundan Singh and Henning Schulzrinne. Peer-to-peer internet telephony using sip. Department of Computer Science, Columbia University.