Project: Distributed Applications

Andrzej Duda

October 26, 2015

1 Principles for all projects

You should work in groups of 4 people. One person must be a leader responsible for the project. Another one (and not more!) will be responsible for documenting the project.

If you are already proficient with some or all protocols required for a project or already have some related experience, join that project, you will be a big help for the others!

Among the multiple technology you will have to use in those projects, one is essential (except for the SIP application): non blocking sockets in Java (java.nio), which is far more effective than solutions based on multi-threads and the traditional blocking sockets. You will find all information to use them in the Chapter 5 of this book: *"TCP/IP sockets in Java: practical guide for programmers", Kenneth L. Calvert, Michael J. Donahoo.*

1.1 Software engineering

You will use git to manage your code[1]. Here are two diagrams to understand basic functions of git (you can find more on the git principles here[2]).

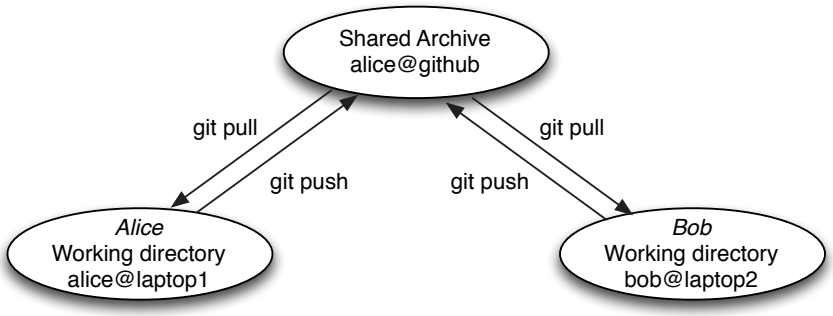

Figure 1: Principles of git.

Your team needs to follow C4.1 - Collective Code Construction Contract[3] and use an account on GitHub[4] (subscribe me to your repository: user `andrzej-duda`).

[1] https://git-scm.com
[2] http://marklodato.github.io/visual-git-guide/index-en.html
[3] http://rfc.zeromq.org/spec:22
[4] https://github.com

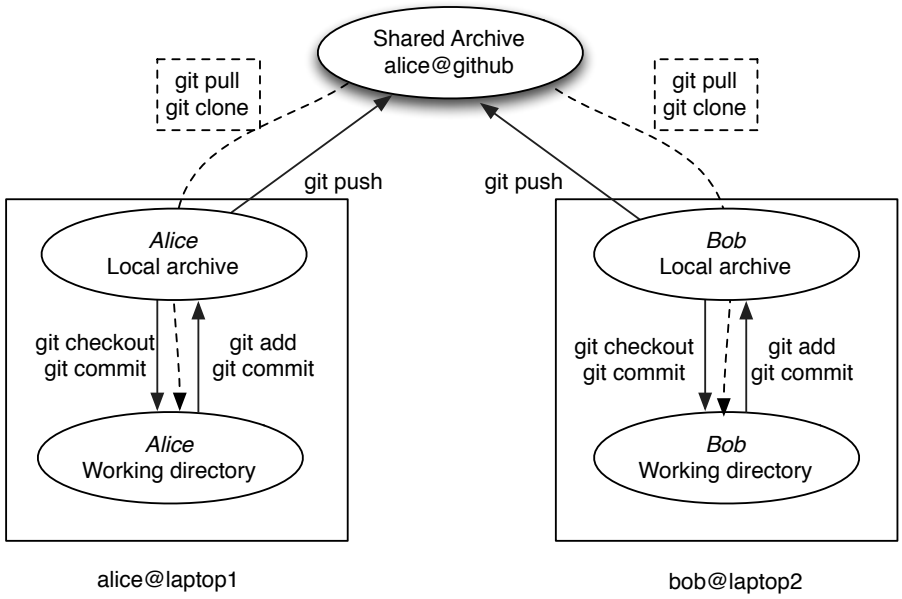

Figure 2: Principles of git.

The recommended GitHub workflow is "social coding"[5]: create a fork, work on your branch, and submit a Pull Request. A colleague reviews the Pull Request, eventually accept it and merge with the master[6].

Always think about a debugging possibility—it is not a mistake to insert the debugging code from the beginning on (the debugging macro/functions should respect the debugging level) and write a log file/execution trace (log4j).

1.2 Project schedule

The project will span 12 weeks and its timeline is as follows:

- weeks 1–2 reading RFCs, man pages, APIs and specifications, understanding the project's goal, discussions, distribute work among team members.
- weeks 3–4 find your tools, clarify the architecture, start to code parts where you are clear about (define data types & classes, make a project skeleton, make/ant files, etc.), start coding simple test programs for things which are new for you.
- weeks 5–6 you should have done simpler parts of the projects and your architecture should be clear now. You are working on the more difficult parts.
- week 7 - intermediate review.
- weeks 8–10 the difficult modules start to work, you try to put all the parts together, minor corrections on the architecture. You write some tests.
- weeks $>$ 10 work hard on the whole... you will need it.

[5] http://spring.io/blog/2010/12/21/social-coding-in-spring-projects

[6] https://www.cocoanetics.com/2012/01/github-fork-fix-pull-request, https://gist.github.com/Chaser324/ce0505fbed06b947d962, https://yangsu.github.io/pull-request-tutorial

- week 12 - final review.

You will need to have at least one meeting per week. Continuous work is expected, you cannot do the project in the last 4 weeks—it will not work.

During week 7, there will be an intermediary review to check your progress: a 30-min slot (15 min presentation + 15 min questions). It will be based on the documentation you will commit in a folder "doc" on your GitHub repository. It will be taken into account in to the final mark.

We need have at least the following documentation on GitHub:

- The testbed you set up to capture with wireshark a basic session with existing applications
- A temporal sequence diagram of the basic session showing the most important phases of the protocols that your application relies on as well as the messages (and their meaningful content) exchanged between the client and the server from the 1st connection of a client until its disconnection. Look at this FTP diagram[7] to see what is expected. You can also detail the specific client/server actions in response to the transmission/reception of each message.
- A figure explaining the designed software architecture (something like Figure 5).
- The final functionalities you expect to have at the end of the project.
- The intermediary steps (increments or small independent applications + testing scenario) you planned before the final product.
- If you use any existing library, a brief explanation of how it works and what you can directly reuse for your application.
- A Gantt diagram of what has been done so far and by whom, and a provisional Gantt diagram to this end of the project. Something as simple as Figure 3 is enough.

Time	Member 1	Member 2	Member 3	Member 4	Time
week1	java (10h)		RFC(5h)		15
	clients tested with "irc.ircity.org"(2h)		Git(2h)	Java(5h)	9
week2	java.nio(7h)		Parser(1h)		8
week3					
week4					
week5					
week6					
week7					
week8					
week9					
week10					
week11					
week12					

Figure 3: Gantt diagram

At the end of the project you should provide:

- the commented, tested, and working (minor bugs are OK but it must be possible to compile and run it without immediate crash) source code.

[7] http://zoo.cs.yale.edu/classes/cs433/cs433-2013-fall/readings/FTP.pdf

- documentation of your project (about 10 pages): architecture, implementation details, result(s) of the test(s) as well as the documentation of the specific parts of each member of your team (who has done what). This will include the enhanced material of the intermediary presentation.

The next section presents the proposed projects.

2 Instant Messenger based on Zeroconf Apple protocol: rewrite a library

If any code is reused (even modified), you should clearly indicate it.

2.1 Project goals

Most chat applications need a server for users to discover each other and exchange messages. However, thanks to the Zeroconf protocol, it is possible to discover on a LAN (without any server) a printer, users that have a chat application etc.

For discovery, you need to implement the following protocols: mDNS (multicast) and DNS-SD.

For instant messaging, the XMPP protocol defines the way to connect to another user to chat, the format of the messages as well as the messages to update your presence status. This aspect is not that difficult as XMPP messages use XML.

The main goal of this project is to **rewrite the library JmDNS** and write an **XMPP Android client** based on the library, able to interact with other XMPP existing applications (e.g. pidgin).

2.2 Project hints

There are three main parts in the project:

1. Service discovery.
2. Support of presence and instant messaging.
3. Graphical client and networking on Android.

To better understand mDNS and DNS-SD, you may "google" a little bit and look at the bibliography below. In parallel, you should "wireshark" message exchanges between 2 XMPP applications that both support Bonjour. You should then be able to code a small piece of code on a PC based on JmDNS able to provide similar messages. Meanwhile, some should begin playing with the Android emulator to discover the Android development environment and the GUI.

The DNS-SD standard is complex so perhaps you might not be able to implement all of it. You should analyze it and find what is essential for this project to work with and start by implementing these parts of the standard. What would be definitely interesting is to see, at the end of the project, the differences between your library and JmDNS.

The graphical client can be developed in parallel or at the end since it will use libraries to do all the real work. This part is the smallest one and you should first concentrate on the first two parts.

To debug your application, it is interesting to interact with tools that support the protocols.

2.3 Resources

- The official mDNS RFC and the mDNS official website
- The official DNS-SD RFC and the DNS-SD official website
- The wikipedia page about the Zero-configuration protocol (Zeroconf) and the Zeroconf official website which includes mDNS and DNS-SD

- Bonjour: The Apple Zeroconf protocol implementation
- XMPP clients that may support the bonjour protocol (Pidgin supports it)
- The XMPP RFC
- The JmDNS library

3 Android SIP phone

If any code is reused (even modified), you should clearly indicate it.

3.1 Project Goals

The Session Initiation Protocol is a text-based application-layer signaling and call control protocol used by applications to establish, maintain, and close multimedia communications.

Its growing success in VoIP companies can be explained by its simplicity and extensibility, which is more close to Internet philosophy than other heavy signaling protocols like H323.

SIP provides a public SIP address to contact a user that looks like an e-mail address (*sip:username@domain_name*) and its main services are user location, capability negotiation, session establishment, and termination. It is used by Instant Messengers, for Internet telephony and video calls or gaming servers.

The goal of this project is to develop an Android SIP phone able to handle simultaneously multiple communications as well as more advanced codecs than GSM or G.711 μlaw. Another additional feature could be to support Push-To-Talk.

With the Jain SIP library, you should be able to quickly (2 weeks) prototype a non graphical SIP client on a PC and have a basic direct SIP interaction (cf. Figure 4) with an existing SIP application like Sjphone or X-lite. Porting it to Android should be straightforward.

Regarding the media part, it may not be as straightforward since audio management is linked to hardware and to the operating system. So, you need to investigate what Android natively supports and if it is enough, or if we should use an external library.

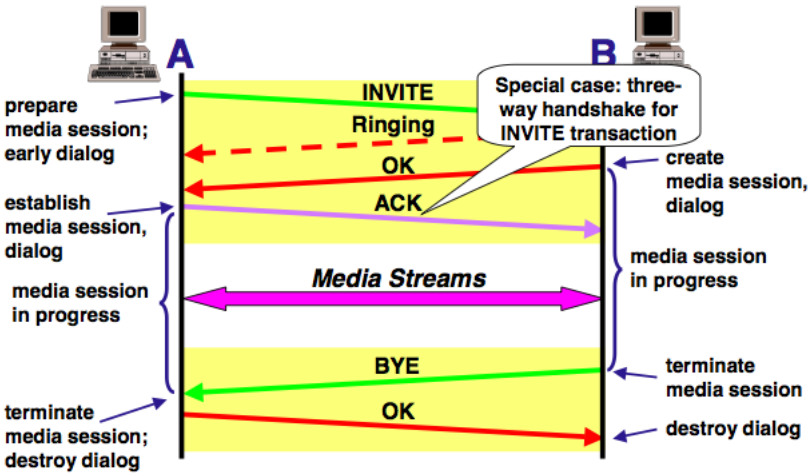

Figure 4: A basic SIP Session

3.2 Architecture hints

Figure 4 illustrates a basic SIP session between two SIP applications called User Agents.

To experiment without having to use a registrar, you can have direct host-to-host calls by using the following URI: "sip:IP_address_of_callee".

Once you call this IP address, the application is able to establish a communication with the end terminal through sending an INVITE message containing a description of the offered session (availability of audio codecs and the UDP port). The phone of the callee rings (RINGING message) to warn him from an incoming call. When he picks up his phone, an OK message is sent back to the caller with the description of the callee available codecs and its UDP port. One more message is sent by the caller (ACK message) and the communication can take place.

To close the SIP session, a BYE message is sent when a user hangs up its phone.

During the communication phase, the RTP (Real Time Protocol) will be used to exchange audio frames.

The description of the session conveyed by the SIP messages are based on the SDP (Session Description Protocol) protocol.

What must be integrated within the application:

- A GUI with minimal functionalities (button to call, answer or close a session, a field to enter the sip address to contact, warning messages for an incoming call, ...): it can be quite rudimentary.
- A Media Manager in charge of the audio communication (RTP).
- A module dedicated to the session description (SDP): negotiation.
- A SIP Manager dedicated to the SIP messages exchanges and the session status evolution (SIP).

First, you should determine the services that the SIP stack provides and at the same time, the way to transmit and receive media over RTP with the appropriate multimedia library.

Do not hesitate to look at the code examples available in the JAIN API sources to begin the User Agent development.

3.3 Software resources

- For the SIP (&SDP) protocol, the Jain SIP library is a reference:
  - The latest stable release
  - A good start for coding a SIP User Agent is the *shootist* and *shootme* examples that are available in jain-sip/src/examples, /simplecallsetup
- For the RTP protocol and audiovideo codecs,
  - for Android applications: the native Android audio library. You can have a first look at supported codecs.
  - for a PC Softphone libjitsi used by Jitsi or by using C libraries and using them in your application thanks to JNI. A perfect tutorial can be found at http://blog.sharedmemory.fr/en/2014/07/07/gsoc-2014-libjitsi-tutorial/
- An existing SIP client to interact with your application: Sjphone but any SIP software can do (X-lite, Linphone). These softphones should have a "Direct SIP" (like SjPhone) profile that allows to test direct calls between hosts without a SIP registrar.

3.3.1 Documentation

- The latest Jain Sip library API
- A tutorial about Jain SIP
- A basic SIP session establishment (more examples in the RFC 3665: Session Initiation Protocol (SIP) Basic Call Flow Examples)
- The RFC 3261: Session Initiation Protocol
- The RFC 2327: Session Description Protocol

4 BitTorrent client

4.1 Project goals

A peer-to-peer (P2P) distributed architecture is composed of participants that make a portion of their resources (such as processing power, disk storage, and network bandwidth) available directly to their peers without intermediate hosts or servers. Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model where only servers provide a service and clients consume.

BitTorrent is a peer-to-peer file sharing protocol used for disseminating files. In this project, a BitTorrent client will be developed.

The client should have the following features:

- Creation of torrent metadata files.
- Multiple parallel downloads.
- Suspend and resume downloads.

The GUI should show the following statistics:

- Current download and upload speeds and estimated time remaining.
- Percentage of each file completed and information on the "pieces" (small parts of the complete file) still needed including the piece number and availability.
- Peer data including IP addresses, the speeds at which you are downloading and uploading to/from them, the port they are running BitTorrent on, and the BitTorrent client they are using.

4.2 Architecture hints

A BitTorrent client is any program that implements the BitTorrent protocol. Each client is capable of preparing, requesting, and transmitting any type of computer file over a network, using the protocol. A peer is any computer running an instance of a client.

To share a file or group of files, a peer first creates a small file called a "torrent" (e.g. MyFile.torrent). This file contains metadata about the files to be shared and about the *tracker*, the computer that coordinates the file distribution.

Peers that want to download a file must first obtain a torrent file for it, and connect to the specified tracker, which tells them from which other peers to download the pieces of the file. The client connects to those peers to obtain various pieces. A piece is identified with a SHA1 hash on its content. After the file is successfully and completely downloaded, the peer is able to shift roles and become an additional seed, helping the remaining peers to receive the entire file.

4.2.1 Bencoding

Bencoding is a way to specify and organize data in a terse format. It supports the following types:

Type	Encoding	Example
byte strings	`<string length encoded in base ten ASCII>:<string data>`	4:spam
integers	`i<integer encoded in base ten ASCII>e`	i3e
lists	`l<bencoded values>e`	l4:spam4:eggse
dictionaries	`d<bencoded string><bencoded element>e`	d3:cow3:moo4:spam4:eggse

4.2.2 Metainfo file structure

All data in a metainfo file is bencoded. The content of a metainfo file (the file "something.torrent") is a bencoded dictionary, containing the mandatory keys listed below. All character string values are UTF-8 encoded.

- **info**: a dictionary that describes the file(s) of the torrent. There are two possible forms: one for the case of a 'single-file' torrent with no directory structure, and one for the case of a 'multi-file' torrent
- **announce**: The announcement URL of the tracker (string)

For the case of the *single-file mode*, the **info** dictionary contains the following mandatory fields:

- **name**: the filename. This is purely advisory (string).
- **length**: length of the file in bytes (integer).
- **piece length**: number of bytes in each piece (integer).
- **pieces**: string consisting of the concatenation of all 20-byte SHA1 hash values, one per piece (byte string, i.e. not urlencoded).

4.3 Getting started

Some hints that may help at the beginning:

- you can look at this document: create your torrent with some existing bittorrent clients.
- Because of the firewall, you probably have to use your own tracker. I recommend opentracker. One of its advantage is that it supports both HTTP and UDP requests. For the latter, the response is not bencoded.
- Implementing bencoding parsing is not needed, you can reuse an existing library.

4.4 Useful links

- The BitTorrent Protocol Specification
  http://www.bittorrent.org/beps/bep_0003.html
- Unofficial Bittorrent Protocol Specification v1.0
  http://wiki.theory.org/BitTorrentSpecification
- BitTorrent Protocol
  http://en.wikipedia.org/wiki/BitTorrent_%28protocol%29
- BitTorrent vocabulary
  http://en.wikipedia.org/wiki/Terminology_of_BitTorrent

5 IRC server and serverless extensions

You will develop a server tool compliant with the RFC 1459 IRC chat protocol. The IRC (Internet Relay Chat) specifies a text based client-server protocol for communicating in real time with several users. A server relays all communication. To enter a session, the user connects to the server and joins a communication channel on which several users exchange text messages. Your tool needs to work with existing networks of IRC servers on the Internet.

In this project, there are two steps:

- develop a centralized IRC server in Java
- extend your code to a serverless version so that all clients act as a server and a client.

For the extension, there is no specification to follow (you need to design it). An idea would be to reuse the JmDNS library already presented in Section 2 to discover the same client/server on the LAN automatically and handle IRC channels and users in a distributed way.

In the following sections, we explain how an IRC server works.

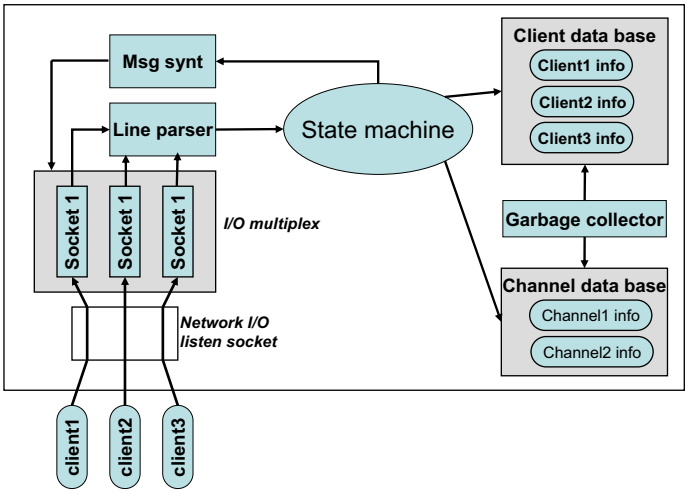

Figure 5: Example IRCd server architecture.

5.1 Project goals

During this software project, a RFC1459 compliant chat server prototype will be developed. The IRC (Internet Chat Relay) standard specifies a text-based client-server communication protocol with many features like: one-to-one, one-to-many, one-to-all communication, modular services, distributed decentralized chat network, and much more. The goal is to develop a prototype accepting standard IRC clients like mIRC (Windows) or ircii (Unix).

The prototype should at least:

- support multiple clients joining multiple channels (#channels also known as channels with modes) and properly route messages between them.
- support one-to-one messages (user1 directly messaging user2).
- on #channels, support at least one changeable mode (for example +t) and the operator mode (adding more modes is trivial once you have done this).

5.2 Architecture hints

The prototype will support the base set of the RFC1459 allowing one-to-one and one-to-many (#channels) communication. The project requires the development of an efficient non-blocking I/O subsystem written in Java managing multiple connected clients in one execution thread (because the information you need is centralized and not per client basis!) using the java.nio package.

Parts of the software should use simple data base operations to handle different channels and users. Basic building blocks of the prototype are shown in Figure 5.

The functional principles of the IRCd (the server) are as follows:

- A listening socket for incoming connections is created.
- For a new client connection, a new socket is created (connection accepted) and a client info structure is created that describes the current state of the client and keeps track of client resources (like client socket, I/O buffers, membership in channels, client modes, etc). The client must register with the server first using two commands NICK and USER (so this can be modeled as a 3 (or 4...) state machine NEW → UNREGISTERED → REGISTERED).
- After the registration phase, the client can send commands (line oriented protocol like COMMAND argument-list NEWLINE to the server that may in turn send a response (depending on the command). The command is interpreted taking the client state into account, e.g. one can only 'speak' in a channel after he has joined that channel.
- The server must route client messages properly, e.g. if two clients join the same channel and one of them speaks, the message must be also sent to the second client. Similarly, if one client performs an action on a channel (like changing channel modes or his nickname), the change message must be routed to the second (third...) channel member.
- The server should periodically check if all clients are still alive (send PING messages) and remove clients that do not respond.
- The IRCd must also maintain a database of existing channels (a limit on the number of per client channels should exist) and for every channel, know its members. A new channel is created if a client joins a channel that did not exist. The first member of a channel automatically gains the channel operator status. The channel is removed (destroyed) after all clients have left it.
- The channel operator status may be given by any operator to another member of the channel and all channel operators are equal (democratic principle with the risk of channels without any operator after a while).

As you see, the essence of the IRCd is to multiplex data between different sockets. Thus, an efficient non-blocking engine must be developed. You should find an abstraction layer for clients and channels.

5.2.1 Resources

- RFC 1459
  http://tools.ietf.org/html/rfc1459
- RFC 2810
  http://tools.ietf.org/html/rfc2810
- RFC 2811
  http://tools.ietf.org/html/rfc2811
- package java.nio
  http://download.oracle.com/javase/1.4.2/docs/api/java/nio/channels/package-summary.html

- the book "TCP/IP sockets in Java: practical guide for programmers, Kenneth L. Calvert,Michael J. Donahoo"
- Chatzilla (Firefox plugin) seems to be quite robust, XChat, or IRC with pidgin