Complex client-server applications

Corrado Leita
corrado_leita@symantec.com

What have we seen so far?

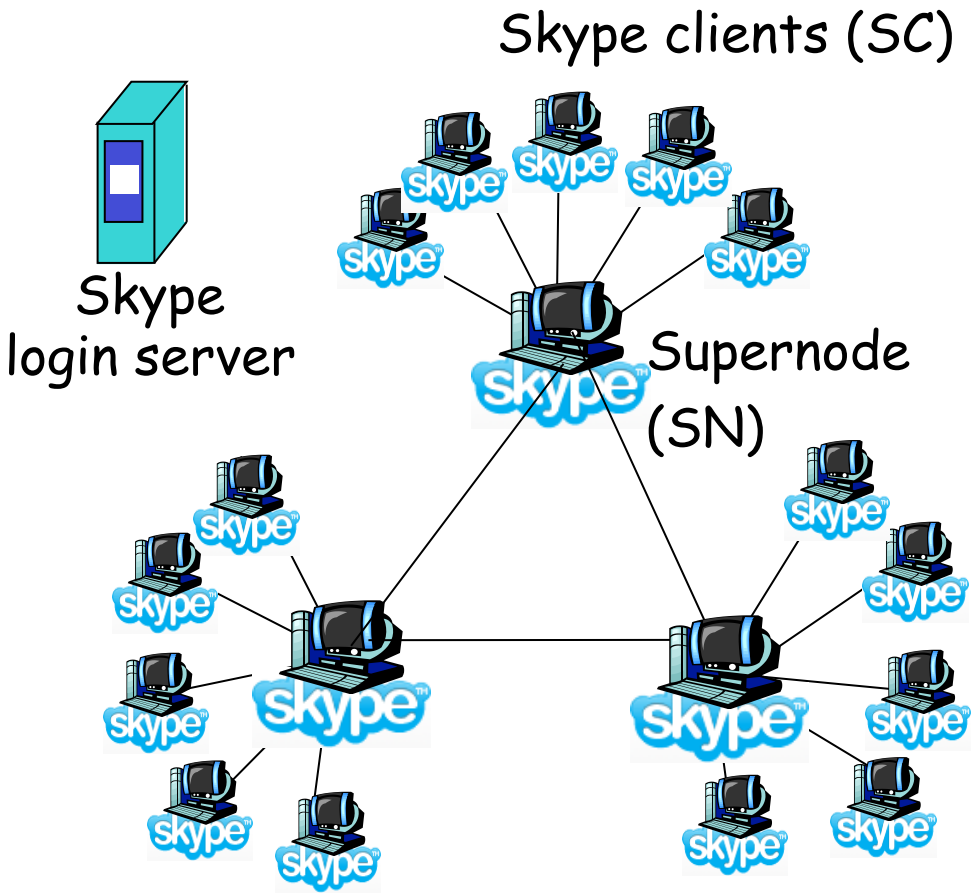
- Different I/O models
  - Blocking I/O
    - Multiprocessed (os.fork())
    - Multithreaded (threading module)
  - Multiplexed I/O
    - Using select.select() to multiplex file descriptors
  - Non-blocking I/O
    - Polling
- Can we apply what we have learned to build complex applications?

P2P Instant Messagenger

- Very didactic example: client and server functionality in a single application
  - Register the instance to a directory server (client)
  - Send messages to other peers (client)
  - Receive messages from other peers (server)
- Pattern actually followed in practice by some protocols
  - IRC (Direct Client-to-Client protocol)
  - Skype
  - …

Skype P2P architecture

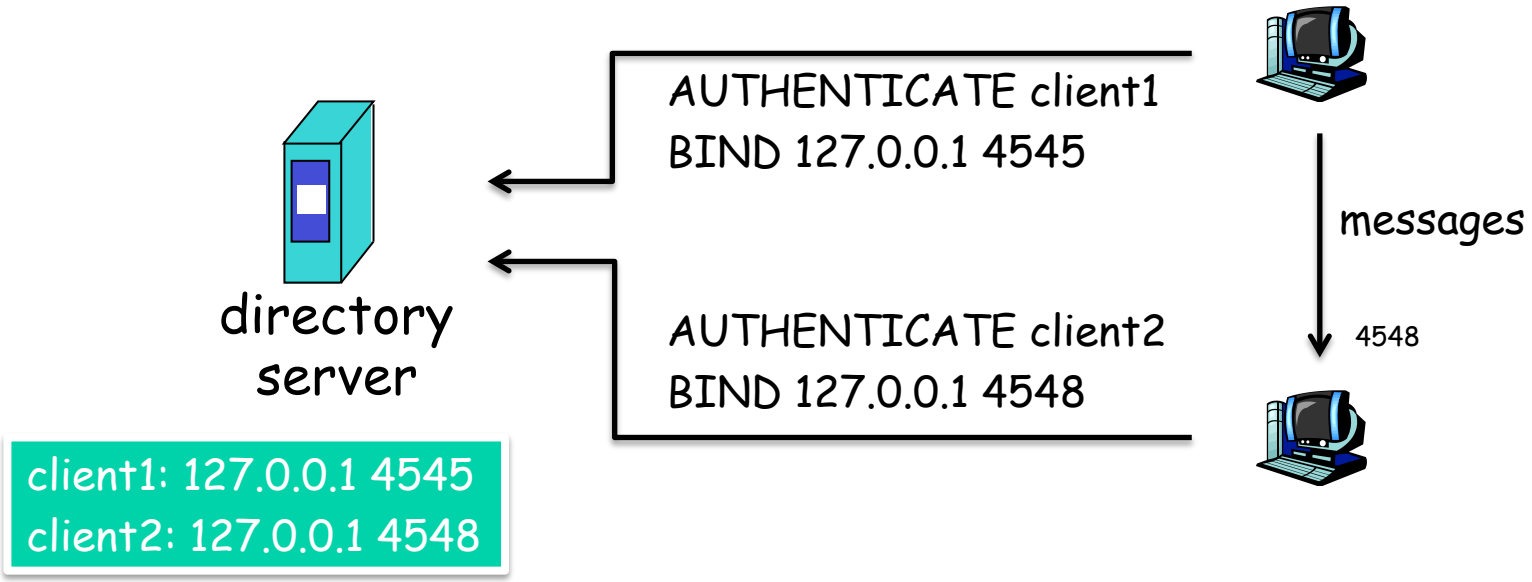
- inherently P2P: pairs of users communicate.
- proprietary application-layer protocol (inferred via reverse engineering)
- hierarchical overlay with SNs
- Index maps usernames to IP addresses; distributed over SNs



Let's make it simpler

- **Directory server** must:
  - maintain an up-to-date list of usernames logged into the system
  - be able to associate to each username the IP address and port on which the user is reachable
  - implement access control
- **Clients** must:
  - be able to authenticate to the directory server
  - bind the address/port they are listening on to their username in the directory server
  - be able to discover peers in the discovery server
  - be able to interact with peers and deliver them messages by means of TCP

eurechat architecture



- client1 connects to the directory server, and discovers client2's endpoint
- client1 connects to 127.0.0.1 4548, and delivers him messages
- client2 can respond to the messages over the same conversation

Protocol primitives

LOGIN: let the directory server know who you are by providing him your username

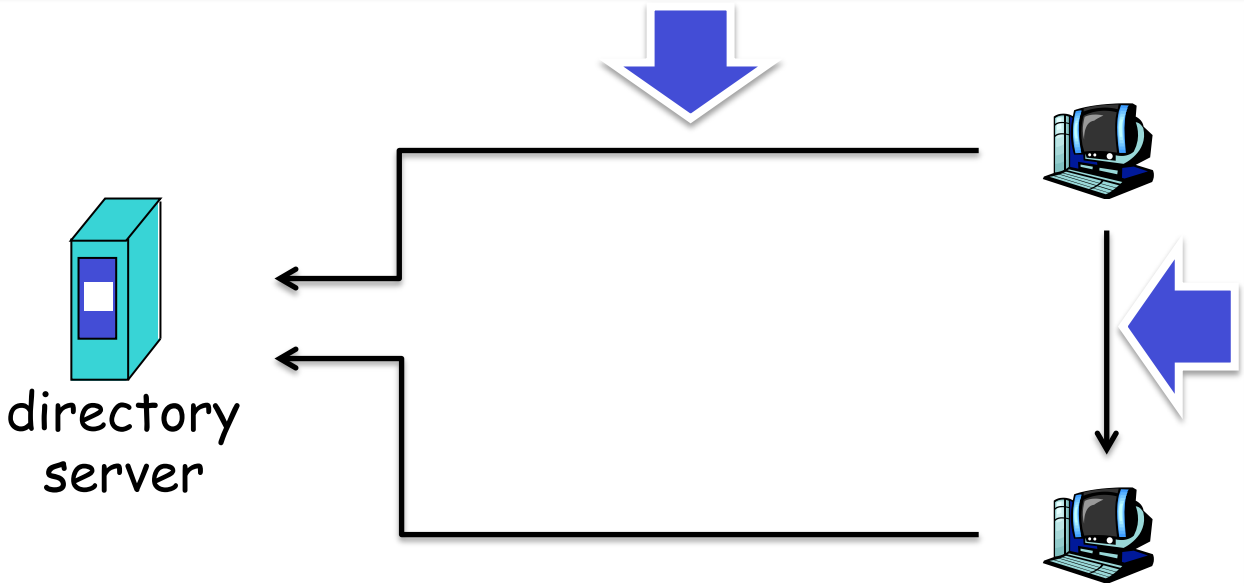
PASS: prove to the directory server that you own the username by authenticating it with a password

BIND: register yourself in the directory with a specific IP address and port in which you can be reached

QUERY: retrieve information from the directory about a specific username or about all the usernames in the system

LEAVE: deregister yourself from the directory service

ACK/ERR/RESULT: messages to ack the correct execution, report errors or results

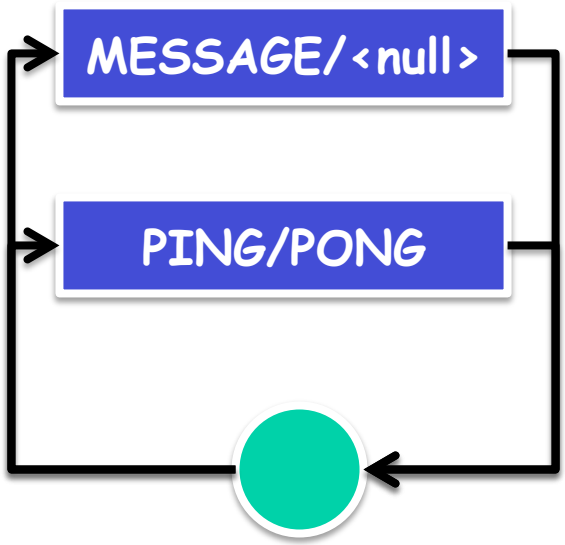


MESSAGE: deliver a chat message to the recipient, letting him know the sender username

PING: request a test of the correct operation of the receiver protocol implementation

PONG: to be sent back in response to a PING message

client-to-client protocol



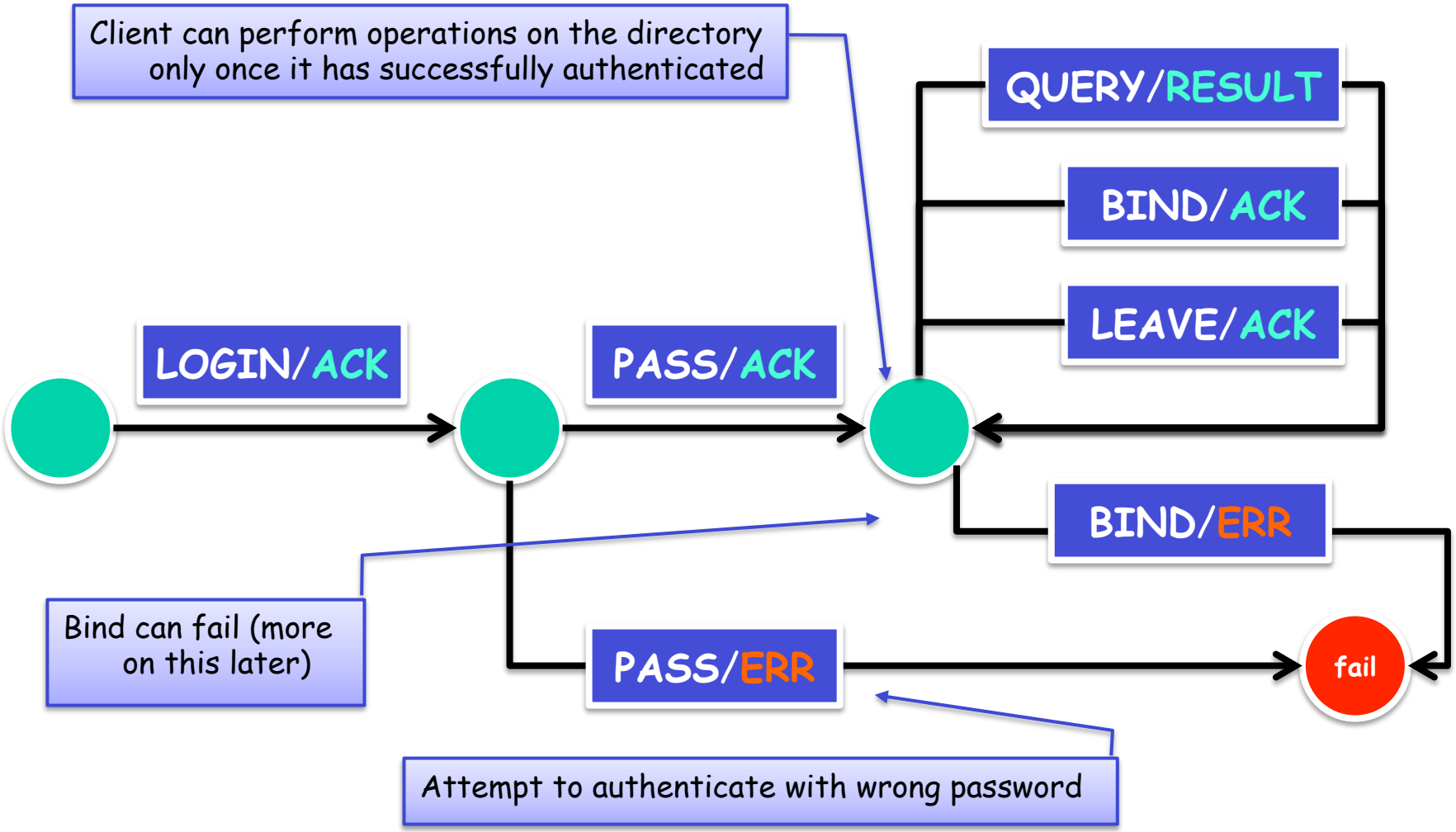
MESSAGE payloads are used to exchange chat messages between the two peers and do not need to be acknowledged. Either the client or the server can send a MESSAGE to the other peer at any time.

PING messages are used to ensure the correct operation of the IM client. As soon as a PING message is received, the peer must immediately respond with a PONG.

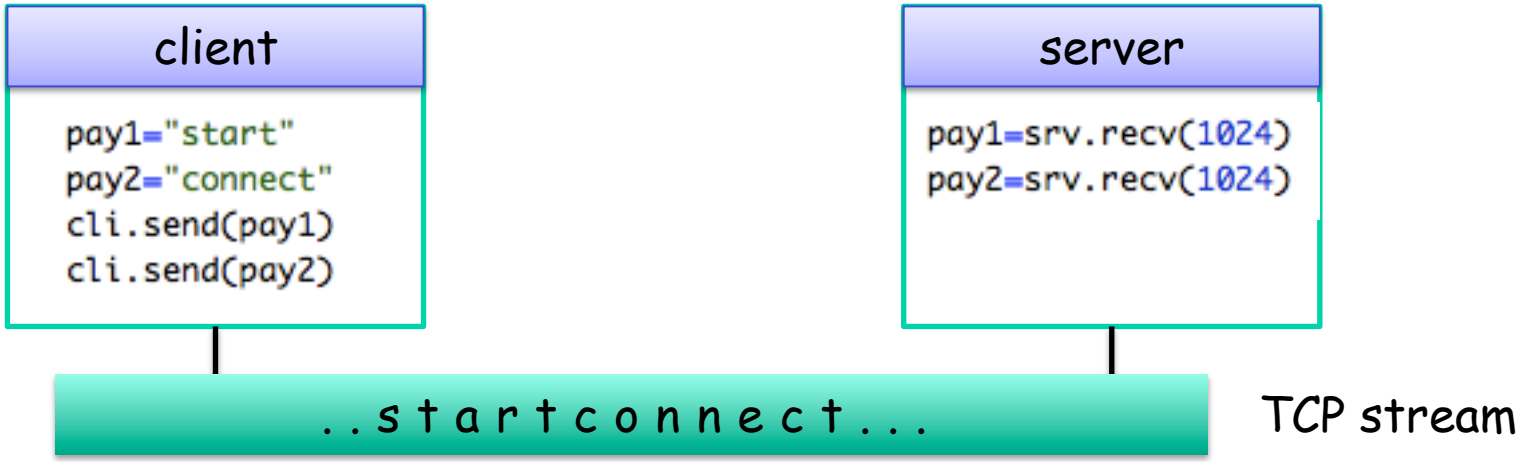
protocol is stateless

read: **<client message>/<server answer>**

client-to-directory protocol



Message syntax



- Reminder from previous lecture: stream sockets do not provide a notion of message boundary
- We need to ensure that delimiting application level messages is always possible

Message syntax

- What do we need?
  - Message type: to distinguish messages with different semantics
  - Message arguments: information that complements the message type with additional semantics
  - Message payload: to deliver arbitrary length and arbitrary content messages to the other endpoint

Message syntax

Plain-text protocols are easier to understand and debug (e.g. looking at packet traces)

Message header. The Message header contains the message type, a number representing the payload length, and an arbitrary number of space-separated arguments. The header is always terminated by a new line character.

Message payload: any sequence of characters of length specified in the header (12 characters in this case).

Important: notice the difference in constraints between arguments and payload. Spacing characters or new line characters are forbidden in the arguments, because they would introduce ambiguities (when does this header end? "MESSAGE 12 send\ner\n"). These ambiguities can be handled through escaping (e.g. HTTP protocol) or through encoding (e.g. SMTP transfers of binary data).

Parsing messages

➔ CODE

eurechat/parser.py

```
class Message:

    def __init__(self,message_type=None,message_args=[],message_payload=""):
        self.type=message_type
        self.args=message_args
        self.payload=message_payload

    def parse(self,buf):
        consumed=0

        match=re.match("(?P<type>\w+) (?P<len>\d+)(?P<args>( \S+)*)\n",buf)
        if match:
            hlen=match.end()
            m_type=match.groupdict()["type"]
            m_len=int(match.groupdict()["len"])
            m_args=[i for i in match.groupdict()["args"].strip().split(" ") if len(i)>0]

            if len(buf)>=hlen+m_len:
                payload=buf[hlen:hlen+m_len]
                consumed=hlen+m_len

                self.type=m_type
                self.args=m_args
                self.payload=payload

        return consumed


    def __str__(self):
        if len(self.args):
            return "%s %d %s\n"%(self.type,len(self.payload)," ".join(map(str,self.args)))+self.payload
        else:
            return "%s %d\n"%(self.type,len(self.payload))+self.payload
```

Every message is wrapped by a class Message, that characterizes each object by its type, its arguments and its payload

When converting the object to string, we follow the syntax previously defined. The string representation of the objet can be directly pushed to a socket.

Parsing messages

```
corrado@daphne code $ python
Python 2.6.7 (r267:88850, Oct  7 2011, 22:28:47)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from eurechat.parsing import Message
>>> m=Message("MESSAGE",["sender"],"Hello world!")
>>> str(m)
'MESSAGE 12 sender\nHello world!'
>>>
```

Parsing messages

- Different ways to build parsers in python
  - Regular expressions (re module): simple and effective for simple examples by leveraging regular expression grouping.
  - pyparsing (http://pyparsing.wikispaces.com/): alternative approach for creating and executing simple grammars

```
from pyparsing import Word, alphas
greet = Word( alphas ) + "," + Word( alphas ) + "!"
hello = "Hello, World!"
print hello, "->", greet.parseString( hello )
```

Regular expression basics

- Basic regexp usage:
  - re.match: try to apply a regular expression pattern to the start of a string returning a match object if it matches.
  - re.search: scan through the string looking for a match to the pattern, returning a match object if one is found.
  - re.findall: return a list of all non-overlapping matches in the string.
- Example:
  - re.match("MESSAGE\s+","MESSAGE 12")!=None
  - re.match("MESSAGE\s+","MESSAGE 12").start() -> 0
  - re.match("MESSAGE\s+","MESSAGE 12").end() -> 9
  - re.match("MESSAGE\s+","MESSAGE 12").span() -> (0,9)
  - re.match("MESSAGE\s+"," MESSAGE 12")==None
  - re.search("MESSAGE\s+"," MESSAGE 12").span() -> (1,10)
  - re.findall("\s"," MESSAGE 12") -> (' ',' ')

Regular expression grouping

- Sometimes we need more information than just whether the RE matched or not. Sometimes we want to dissect a string by writing a RE divided into several subgroups which match different components of interest.
- Groups:
  - marked by the '(' , ')' metacharacters
  - named through the '(P<name>) convention

```
>>> re.match("(?P<g1>Hello) ?(?P<g2>World)(?P<g3>!+)","Hello World!!!").groupdict()
{'g3': '!!!', 'g2': 'World', 'g1': 'Hello'}
>>>
```

Going back to our parser

```
re.match("(?P<type>\w+) (?P<len>\d+)(?P<args>( \S+)*)\n",buf)
```

- Now we can fully understand the regular expression.
  - type: one or more alphanumeric character
  - len: one or more decimal digit
  - args: zero or more non-whitespace characters separated by a space

Parsing messages

→ CODE

eurechat/parser.py

```
class Message:

    def __init__(self,message_type=None,message_args=[],message_payload=""):
        self.type=message_type
        self.args=message_args
        self.payload=message_payload

    def parse(self,buf):
        consumed=0

        match=re.match("(?P<type>\w+) (?P<len>\d+)(?P<args>( \S+)*)\n",buf)
        if match:
            hlen=match.end()
            m_type=match.groupdict()["type"]
            m_len=int(match.groupdict()["len"])
            m_args=[i for i in match.groupdict()["args"].strip().split(" ") if len(i)>0]

            if len(buf)>=hlen+m_len:
                payload=buf[hlen:hlen+m_len]
                consumed=hlen+m_len

                self.type=m_type
                self.args=m_args
                self.payload=payload

        return consumed


    def __str__(self):
        if len(self.args):
            return "%s %d %s\n"%(self.type,len(self.payload)," ".join(map(str,self.args)))+self.payload
        else:
            return "%s %d\n"%(self.type,len(self.payload))+self.payload
```

Notice: the parse function always returns the number of bytes that have been consumed to parse the buffer (to handle buffering of multiple messages)

Parsing messages

```
def parse(buf):
    m=Message()
    consumed=m.parse(buf)
    toparse=buf[consumed:]

    return (toparse,m) if consumed>0 else (toparse,None)


def parse_all(buf):
    parsed=[]
    toparse=buf

    while len(toparse)>0:
        toparse,m=parse(toparse)

        if m==None and len(toparse)>0:
            break
        elif m!=None:
            parsed.append(m)

    return toparse,parsed
```

Simply returns a parsed message (if at least one full message was present in the buffer) and the remaining portion of the buffer that is still unparsed.

Scan through the whole buffer and try to parse as many messages as possible.

→ CODE

eurechat/parser.py

Parsing messages

```
Python 2.6.7 (r267:88850, Oct  7 2011, 22:28:47)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from eurechat.parsing import parse, parse_all,Message
>>> m=Message("LOGIN",["username"])
>>> n=Message("PASS",["password"])
>>> s=str(m)+str(n)
>>> s
'LOGIN 0 username\nPASS 0 password\n'
>>> parse(s)
('PASS 0 password\n', Message('LOGIN',['username'],''))
>>> parse_all(s)
('', [Message('LOGIN',['username'],''), Message('PASS',['password'],'')])
>>> parse_all(s+"TRUNCATED...")
('TRUNCATED...', [Message('LOGIN',['username'],''), Message('PASS',['password'],'')])
>>>
```

Simplifying socket interaction

```
class ProtocolWrapper:

def __init__(self,socket,addrinfo):
        self.__sock=socket
        self.__address=addrinfo

        self.__buffer=""
        self.__logger=logging.getLogger("endpoint.%s:%d"%self.__address)
        self.__sock.settimeout(30)

    def recv(self):
        msg=None

        while msg==None:
            try:
                pay=self.__sock.recv(1024)
                self.__buffer+=pay
                if not pay:     break
            except socket.error,e:
                self.__logger.error("receive error: %s"%str(e))
                break

            self.__buffer,msg=p.parse(self.__buffer)

        return msg
```

Wrap an already established socket connection (provided in the constructor)

Create a buffer to store incomplete payloads

Incrementally receive data and store it into the buffer.

Use the previously introduced parser to try to pull a full message out of the buffer. Continue to loop to the recv call until a message has been reconstructed.

→ CODE

eurechat/protocol.py

Simplifying socket interaction

```
class ProtocolWrapper:

def __init__(self,socket,addrinfo):
        self.__sock=socket
        self.__address=addrinfo

        self.__buffer=""
        self.__logger=logging.getLogger("endpoint.%s:%d"%self.__address)
        self.__sock.settimeout(30)

    def recv(self):
        msg=None

        while msg==None:
            try:
                pay=self.__sock.recv(1024)
                self.__buffer+=pay
                if not pay:     break
            except socket.error,e:
                self.__logger.error("receive error: %s"%str(e))
                break

            self.__buffer,msg=p.parse(self.__buffer)

        return msg
```

Notice the use of socket.settimeout: we want to avoid blocking a thread forever while waiting for data. After 30 seconds, any blocking call will timeout.

recv will:

1) return a message object if a full message is received
2) return None if the client closed connection
3) raise a socket.timeout exception if the timeout was triggered while receiving

→ CODE

eurechat/protocol.py

Simplifying socket interaction

```
def send(self,message_type,message_args=[],message_payload=""):
    m=p.Message(message_type,message_args,message_payload)

    try:
        self.__sock.sendall(str(m))
        return True
    except socket.error,e:
        self.__logger.error("send error: %s"%str(e))
        return False

def close(self,failure=None):
    if failure!=None:
        self.__logger.debug("failure: %s"%failure)
        self.send(p.T_ERR,[],failure)

    self.__logger.debug("closing connection")
    self.__sock.close()
```

Notice the use of sendall instead of the standard send. Sendall is a shortcut, that continues to invoke send until all data has been pushed to the send buffer.

A send error may still happen, for instance if the stream has been closed. The send function will return False in this case.

→ CODE

eurechat/protocol.py

Simplifying socket interaction

```
def send(self,message_type,message_args=[],message_payload=""):
    m=p.Message(message_type,message_args,message_payload)

    try:
        self.__sock.sendall(str(m))
        return True
    except socket.error,e:
        self.__logger.error("send error: %s"%str(e))
        return False

def close(self,failure=None):
    if failure!=None:
        self.__logger.debug("failure: %s"%failure)
        self.send(p.T_ERR,[],failure)

    self.__logger.debug("closing connection")
    self.__sock.close()
```

When shutting down the connection, we can provide a failure message that will be wrapped in an ERR message, letting the client know about a failure

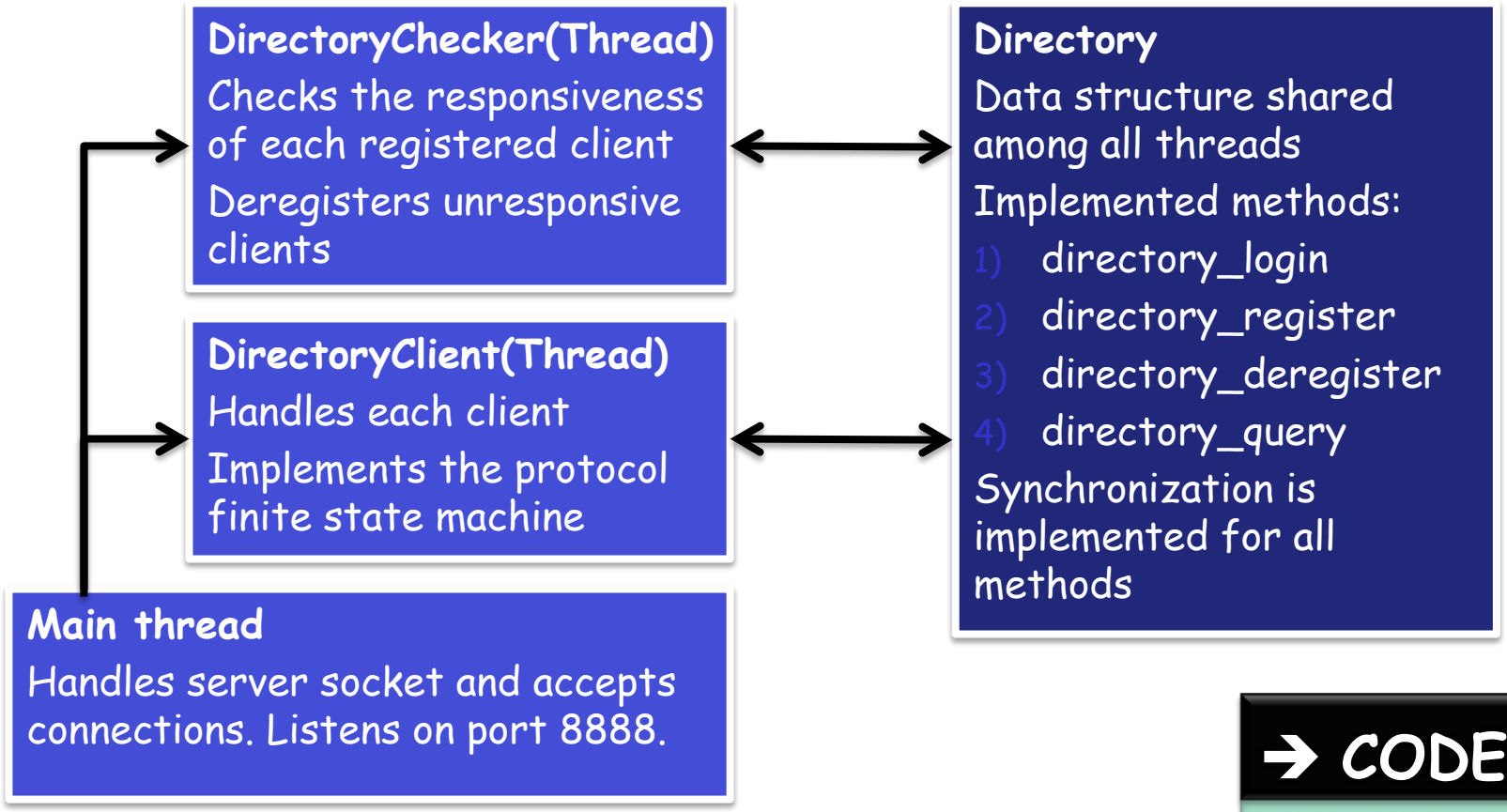
→ CODE

eurechat/protocol.py

A few words on logging

- Logging is important!
- Logging API is provided as a standard library module in python, and it is used extensively by a variety of third party modules
- Hierarchical organization
  - logging.getLogger("eurechat.parsing") returns the logger for the parsing functionality of eurechat. Changing the configuration of the "eurechat" will apply to all its subloggers
  - logging.getLogger("someLogger") always returns a reference to the same logger within the same python process
  - Different logging levels: DEBUG, INFO, WARNING, ERROR can be set for any point of the logging hierarchy

Directory server architecture



→ CODE

directory.py

Protocol finite state machine (authentication)

Never assume that a send is always successful. Things may go wrong in real life!

Handle corner cases correctly, and ensure that the received message is indeed what was expected. Terminate connection otherwise

```
msg=self.__protocol.recv()
if msg==None or msg.type!=p.T_USER or len(msg.args)!=1: return self.__protocol.close("a 'USER <username>' command was expected!")
self.username=msg.args[0]
if not self.__protocol.send(p.T_ACK,[],"hi %s, authentication required"%self.username): return self.__protocol.close()

#get the password
msg=self.__protocol.recv()
if msg==None or msg.type!=p.T_PASS or len(msg.args)!=1: return self.__protocol.close("a 'PASS <password>' command was expected!")
self.password=msg.args[0]

if self.__directory.directory_login(self.username, self.password):
    if not self.__protocol.send(p.T_ACK,[],"successfully authenticated"): return self.__protocol.close()
else:
    return self.__protocol.close("authentication failed")
```

Rely on a directory method for the authentication. Directory returns always true in this example≈

Protocol finite state machine (after authentication)

```
while True:
    msg=self.__protocol.recv()
    if msg==None:   return self.__protocol.close() #connection was closed by client

    if msg.type==p.T_BIND and len(msg.args)==2:
        self.bind_address=msg.args[0]
        self.bind_port=int(msg.args[1])

        if self.__port_test():
            self.__directory.directory_register(self.username, self.bind_address, self.bind_port)
            if not self.__protocol.send(p.T_ACK,[],"bound successfully to %s:%d"%(self.bind_address,self.bind_port)): return self.__protocol.close()
        else:
            return self.__protocol.close("invalid bind notification")

    elif msg.type==p.T_QUERY and len(msg.args)<=1:
        username=msg.args[0] if len(msg.args)==1 else None
        result=self.__directory.directory_query(username)
        payload="\n".join(["%s,%s,%d"%i for i in result])

        if not self.__protocol.send(p.T_RESULT,[],payload): self.__protocol.close()
    elif msg.type==p.T_LEAVE and len(msg.args)==0:
        self.__directory.directory_deregister(self.username)
        if not self.__protocol.send(p.T_ACK,[],"deregistered from directory"): return self.__protocol.close()
    else:
        return self.__protocol.close("I did not understand the message %s"%msg.type)
```

Avoid having incorrect information in the directory: always verify that a client is listening on the port/address provided in the BIND request before proceeding.

A query returns a payload composed of lines:
<username>,<address>,<payload>
If a user is provided as argument, only information on that user is returned.

A well-behaving client should always deregister itself upon exit….

DirectoryChecker

- Avoid having stale information in the directory (clients could be crashing or could be badly implemented… ☺)
- Separate thread continuously querying the directory for fresh information
- For each registered user:
  - fetch address/port information
  - try to connect
  - send a PING message and wait for PONG
  - if any error encountered, deregister the user

DirectoryChecker

```
while True:
    time.sleep(DirectoryChecker.LOOP_WAIT)

    users=[i[0] for i in self.__directory.directory_query()]
    self.__logger.debug("%d users are active"%len(users))

    for username in users:
        res=self.__directory.directory_query(username)
        username,address,port=res[0] if len(res)==1 else (username,None,None)

        if address!=None and port!=None:
            try:
                self.__logger.debug("connecting to %s:%d"%(address,port))
                sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
                sock.connect((address,port))

                proto=ProtocolWrapper(sock,(address,port))
                self.__logger.debug("sending ping")
                proto.send(p.T_PING)
                msg=proto.recv()

                if msg!=None and msg.type==p.T_PONG:
                    self.__logger.info("USER %s OK"%username)
                    proto.close()
                else:
                    raise Exception, "no PONG received"

            except Exception,e:
                self.__logger.error("USER %s ERROR (%s)"%(username,str(e)))
                self.__directory.directory_deregister(username)
```

It actually implements polling: it continues to loop over the directory content, sleeping X seconds among each iteration.

Create a socket, and then wrap it using the ProtocolWrapper

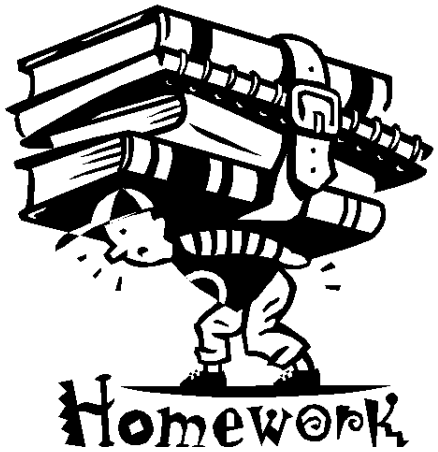
Any exception generated in the test deregisters the user.

Quick test

```
corrado@daphne code $ nc localhost 8888
USER 0 corrado
ACK 35
hi corrado, authentication requiredPASS 0 mypass
ACK 26
successfully authenticatedBIND 0 127.0.0.1 5555
ACK 36
bound successfully to 127.0.0.1:5555QUERY 0
RESULT 22
corrado,127.0.0.1,5555
```

- Run a simple listening process on a separate terminal window:
  - while true; do nc -l 5555; done

HOMEWORK



About the homework

- Write a client able to register itself to the directory server, and able to interact with other clients
  - Implement authentication with directory server
  - Implement client-to-client protocol
- Chatting will never be as fun as while using the application you implemented! ☺

When?

- Dec 5: Homework will be officially enabled (this afternoon)
- Jan 9: Lecture on asynchronous programming with fast Q/A session at the end (a "part 2" will be added to the homework for those who already finished)
- Jan 23: **Deadline** for homework completion
- Jan 30: final lecture with discussion on the homework results

Where?

If you already have a softdev username, **DO NOT** create a new one!

This is the login page for the Introduction to Networks course homework.
This page will allow you to choose your own username and receive a temporary password that you will use to login into the system.
IMPORTANT!! If you are already registered to the SoftDev course, you **must not** create a new login!

First name:

Last name:

Email address:

Username:

Registration token: letmein (it was given to you in the class)

Submit

http://bit.ly/netwi_homework (or http://193.55.112.69/registration_nw)

How?

- If you don't have yet a softdev username, use the web interface to create one and get a temporary password
- ssh to 193.55.112.69
- You will be asked to set a new password
- Read the README.txt!!!
- You will find more information on the challenge in challenges/netw1

Main things to know

- You will find a directory named challenges, with one subdirectory per assignment. This course assignments will start with "netw"
- You can check the game progress using the "game" command
- You can submit a solution by using the "submit" command
  - submit netw1 myarchive.tar.gz
  - submit -I
- Everything you do and everything you store on this machine can be monitored.
  - Do not abuse the system
  - If you find a bug in the system, report it as soon as possible. Do NOT take advantage of it.
  - If you cause a DoS or bother users you will face our anger.
- Cheating is forbidden.
- Copying the solution from other students is easy to detect, stupid and even more forbidden.

Submission

- For all the networking challenges, you are required to submit a TAR archive containing
  - All the source code required to run your client
  - A file named TOKEN and placed in the root of the archive file containing the final token obtained from the execution of the challenge as well as your username:
    - <username> <token>

Questions??