Application-Level Communications

Instructor: Dr. Tongping Liu

Outline

- Fundamentals
  - Client/Server communication protocols
    - Request vs. Request-reply vs. Request-reply-acknowledge
  - Invocation semantics
    - Exact once vs. at least once vs. at most once
  - Communication types
    - Transient vs. persistent
    - Synchronous vs. asynchronous
- Models for application communications
  - RPC: remote procedure call
  - Message-oriented communication
  - Stream-Oriented communication
  - Multicast communication

What are the Problems?

- Process A want to send process B a message
  - "Tongping Liu, Assistant Professor, CS Dept UTSA, x5550"
- Application-level agreement
  - Which part is what (name, title, contact)
- Network: actual message transmission
  - Issues with addressing, performance, scalability, reliability, security

C/S Comm.: Request Protocol (R)

- No value to be returned
- Client requires no confirmation
- The client may proceed immediately after the request message is sent

send(...)  request  receive(...)  exec op;
Client      Server
Continue execution
C/S Comm.: Request-Reply Protocol (RR)

- If client expects results from a server
  - Client requests service execution from server through a request message, and
  - Delivery of service results in a reply message
- Most client-server interactions are built on RR protocol

Request-Reply-Acknowledge Protocol (RRA)

- In addition to RR protocol, client sends acknowledgement after it received reply
- Acknowledgement (with request ID) sent asynchronously

What may go wrong?

- Request lost
- Reply lost
- Server down

Traditional Failure Handling in RR

- Client side: resend requests
  - Client keeps local copy of requests
  - If timeout and no reply received, it resends request
  - After N attempts, assumes that server has failed
- Server side: receives a request multiple times
  - Re-execute & resend results; or
  - Store requests & results: filter duplicated requests and re-transmit results

How long should servers keep the requests & results?
Should Servers Re-Do Operations?

- **Idempotent** operations: which can be performed repeatedly with the same effect.
  - suppose $x$ is input message $\rightarrow f(f(x)) = f(x)$
  - No state needs to maintain on the server

- Are the following operations idempotent?
  - HTTP GET …
  - UNIX file operations: read, write etc.
    - yes
    - NO

Server Invocation Semantics in RR

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request message</td>
<td>Duplicate filtering</td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Server Invocation Semantics (cont.)

- **Maybe**: if no reply, the client does not know if method was executed or not
- **At least once**: will guarantee that invocation be carried out at least once, but possibly more
- **At most once**: Will guarantee that invocation be carried out at most once, but possibly none at all
  - Detect duplicated requests with sequence numbers
  - No guarantees: When a server crashes, the client gets no help and no promises about what happened

Local invocation: **exactly once** - ideal case

http://lass.cs.umass.edu/~shenoy/courses/spring13/lectures/notes/677_lect09.pdf

Types of Communications

- **Asynchronous communication**
  - Sender continues immediately after it has submitted the request (unblocked, need a local buffer at the sender)
- **Synchronous communication**
  - Sender blocks until the sender receives an OK to continue; where the OK may come?
Types of Communications

Persistent vs. Transient
- **Persistent**: A message is stored at a communication server as long as it takes to deliver it (e.g., e-mail).
- **Transient**: A message is stored as long as sender and receiver are working at the same time (TCP, UDP, IP routing).

Asynchronous
- **Persistent**
  - Message-oriented middleware (MOM)
  - Processes send each other messages (queued)
  - Sender does not need to wait for immediate reply
  - Middleware often ensures fault tolerance

- **Transient**
  - Client/Server, RPC, TCP
  - Client and server have to be active at time of communication
  - Client issues request and blocks until it receives reply
  - Server essentially waits only for incoming requests, and subsequently processes them

Drawbacks of synchronous communication
- Client cannot do any other work while waiting for reply
- Failures have to be handled immediately; the client is waiting
- The model may simply not be appropriate (mail, news)

Persistence and Synchronization 1
- a) Persistent asynchronous communication (e.g., email)
- b) Persistent synchronous communication

Persistence and Synchronization 2
- c) Transient asynchronous communication (e.g., UDP)
- d) Receipt-based transient synchronous communication (e.g., TCP)
Persistence and Synchronization

e) Delivery-based transient synchronous communication at message delivery (e.g., asynchronous RPC)

f) Response-based transient synchronous communication (RPC)

Outline

- Fundamentals
  - Client/Server communication protocols
  - Request vs. Request-reply vs. Request-reply-acknowledge
  - Invocation semantics
    - Exact once vs. at least once vs. at most once
  - Communication types
    - Transient vs. persistent
    - Synchronous vs. asynchronous
- Models for application communications
  - RPC: remote procedure call
  - Message-oriented communication
  - Stream-Oriented communication
  - Multicast communication

High-Level Communication Models

- RPC: remote procedure call
- Message-oriented communication
- Stream-Oriented communication
- Multicast communication

Remote Procedure Calls

- Network communication:
  - Send and receive are not transparent
  - Application developers are good at "procedure call" model
- In 1984, Birrell and Nelson allows programs to call procedures located on remote machines (RPC)
  - RPC is integrated into programming languages
  - Like centralized computing, by allowing remote services to be called as procedures
  - Issues: How to pass parameters, semantics in face of errors, failures, etc...

RPC is a high-level model for client-server communication
Review of Local Procedure Call

- Consider a call in C: `result = add(para_x, para_y)`
- Caller pushes parameters onto the stack: in one process
- Stack before (a) and while (b) the called procedure is active is shown below

Parameters Passing in Local Calls

- **Call-by-value**: the parameter value copied to the stack. Modifications do not affect the calling side
- **Call-by-reference**: the address of the parameter is pushed onto the stack (for example, pointers in C; `obj ref` in Java). Any modification affects the variable at the calling side.
- **Call-by-copy/restore**: Copy the variable first to the stack (as in call-by-value), and then copy back after the call, overwriting the caller's original value.
  - In many cases, the same behavior as “call-by-reference”

Remote Procedure Call (RPC)

- myAdd()
- mySub()
- myMax()
- myMin()

- magicAdd()
- magicSub()
- magicMax()
- magicMin()

Client would like to use its own functions as well as the ones in the server. How?

Stubs at Client and Server Sides

- Make remote procedure call look like local call
- So that the client can call `myAdd()` and `magicAdd()` in the same way
  
  \[
  \begin{align*}
  \text{result1} &= \text{myAdd}(\text{para}_x, \text{para}_y) ; \\
  \text{result2} &= \text{magicAdd}(\text{para}_x, \text{para}_y) ; \\
  \end{align*}
  \]

Client

Server

Client stub for magicAdd() packs the parameters into a message and call OS (through send primitive). It will then block itself (through receive primitive).

client stub for magicAdd() unpacks the message and returns result to the client

The server stub for magicAdd() unpacks the parameters from the message and calls the local procedure in a usual way. When completes, the server stub will pack the result and return it to the client.
RPC Steps

A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub.
2. The client stub builds a message and calls the local operating system.
3. The client’s OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server’s OS sends the message to the client’s OS.
9. The client’s OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Sounds simple but there are several issues...

RPC Mechanism

Client computer

Server computer

Client machine

1. Client call to procedure

Server machine

6. Stub makes local call to “add”

2. Stub builds message

proc. “add” int int

3. Message is sent across the network

Server OS

4. Server OS hands message to server stub

5. Stub unpacks message

Client OS

pack value parameters into a message and send it to the server. Would it be that easy?

Big Endian and Little Endian

- Big Endian:
  - Store the most significant byte in the smallest address

- Little Endian:
  - Store the least significant byte in the smallest address.

0x 90AB12CD

Big Endian

Little Endian
Problem 1: different data representations

A process on an Intel machine sends a message of an integer and four-character string (“5, JILL”) to another process on a Sun SPARC machine.

(a) original message on Intel (x86, Little Endian)
(b) receipt message on SPARC (Big Endian) “5000, JILL”
(c) simple reverse: message after converted “5, LLIJ”

The number is reversed by the string is not 29.

Problem 2: Un/Marshaling

More than just wrapping parameters into a message

Client and server machines may have different data representations (think of byte ordering)

Client and server have to agree on a standard representation (e.g., external data representation (XDR))

How are basic data values represented (integers, floats, characters)

How are complex data values represented (arrays, unions)

Client and server need to properly interpret messages, and transform them into machine-dependent representations.

Approaches for Exchanging Information

How can clients make servers on different machines understand them?

With listener’s language

- English, Chinese, Indian

A common external data representation

- Language defined representation
  - e.g., CORBA CDR

External self-descriptive data representation

- e.g., XML, Web Services

Other Problems in Parameter Passing

Passing reference parameters

- Pointers, and in general, reference parameters are passed with considerable difficulty

Solutions

- Forbid reference parameters
- Copy the entire data structure (e.g., an entire array may be sent if the size is known). In the case of the server input parameter, it does not need to be copied back.

How to handle complex data structures (e.g. general graphs)?
Parameter Passing Semantics in RPC

- RPC parameter passing:
  - RPC assumes copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
  - RPC assumes all data that is to be operated on is passed by parameters. Excludes passing references to data.

- Conclusion: full access transparency cannot be realized.

- Observation: If we introduce a remote reference, access transparency can be enhanced:
  - Remote reference offers unified access to remote data.
  - Remote references can be passed as parameter in RPCs.

RPC in Practice

- Essence: developers concentrate on client- & server-specific codes, while RPC (generators / libraries) do the rest (actual communication).
- Client stub and server stub communicate over the network to perform RPC.

Generate Stubs

- Hiding a remote procedure call requires that the caller and callee agree on interface:
  - the format of the messages
  - the representation of simple data structures (no complicated data)
  - the transport layer protocol (Connection-oriented or not)

- Using Interface Definition Language (IDL) to specify an interface: services provided by servers.
  - Compile IDL to generate client & server stubs.

Binding a Client to a Server

- Registration of a server makes it possible for a client to locate the server and bind to it.
- Locate the server’s machine → locate server.
Binding

- **Server**
  - Export server interface during initialization
  - Send name, version no, unique identifier, handle (address) to binder
- **Client**
  - First RPC: send message to binder to import server interface
  - Binder: check to see if server has exported interface
- **Performance issues**
  - Exporting and importing incurs overhead
  - Binder can be a bottleneck (Use multiple binders)
  - Binder can do load balancing

Lightweight RPCs

- Many RPCs occur between client and server on same machine
  - Need to optimize RPCs for this special case => use a lightweight RPC mechanism (LRPC)
- Server exports interface to remote procedures
- Client on same machine imports interface
- OS kernel creates data structures including an argument stack shared between **Server** and **Client**

Lightweight RPCs (cont.)

- **RPC execution**
  - Push arguments onto stack
  - Trap to kernel
  - Kernel changes mem map of client to server address space
  - Server thread executes procedure (OS upcall)
  - Thread traps to kernel upon completion
  - Kernel changes the address space back and returns control to client
Other RPC Models

- Asynchronous RPC
  - Request-reply behavior often not needed
  - Server can reply as soon as request is received and execute procedure later

- Deferred-synchronous RPC
  - Use two asynchronous RPCs
  - Client needs a reply but can't wait for it; server sends reply via another asynchronous RPC

- One-way RPC
  - Client does not even wait for an ACK from the server
  - Limitation: reliability not guaranteed (Client does not know if procedure was executed by the server).

Traditional RPC vs. Asynchronous RPC

Try to get rid of the strict request-reply behavior, and let the client continue without waiting for an answer from the server.

Deferred Synchronous RPC

- Client can also do a (non)blocking poll at the server to see whether results are available.
- Through two asynchronous RPCs

One-Way RPC

- Client does not know if the request is accepted or not (ch 8 fault tolerance)
RPC Limitations

- Parameters passed by values only and pointer values are possibly not allowed.

- Speed: remote procedure calling (and return) time (i.e., overheads) can be significantly (1 - 3 orders of magnitude) slower than that for local procedure.
  - This may affect real-time design and the programmer should be aware of its impact.

Failure: RPC is more vulnerable to failure (since it involves communication system, another machine and another process).

- The programmer should be aware of the call semantics, i.e. programs that make use of RPC must have the capability of handling errors that cannot occur in local procedure calls.

Design Issues

- Exception handling
  - Necessary because of possibility of network and nodes failures;
  - RPC uses return value to indicate errors;

- Transparency
  - Syntactic → achievable, exactly the same syntax as a local procedure call;
  - Semantic → impossible because of RPC limitation: failure (similar but not exactly the same);

Outline

- Fundamentals
  - Client/Server communication protocols
    - Request vs. Request-reply vs. Request-reply-acknowledge
  - Invocation semantics
    - Exact once vs. at least once vs. at most once
  - Communication types
    - Transient vs. persistent
    - Synchronous vs. asynchronous

- Models for application communications
  - RPC: remote procedure call (inherent synchronous)
  - Message-oriented communication
  - Stream-Oriented communication
  - Multicast communication
Stream-Oriented Communication

- Applications
  - Video-conference
  - Online multimedia player/viewer (YouTube, CNN Live etc.)
- Support for continuous media
- Streams in distributed systems
- Stream management

Continuous Media

- Observation: All discussed communication facilities are based on a discrete or time-independent exchanging of information

- Continuous media: Characterized by the fact that values are time dependent:
  - Audio
  - Video
  - Animations
  - Sensor data (temperature, pressure, etc.)

Transmission of Continuous Media

- Different timing guarantees: 3 types of transmission
  - Asynchronous: no restrictions with respect to when data is to be delivered
  - Synchronous: define a maximum end-to-end delay for individual data packets
  - Isochronous: define a maximum and minimum end-to-end delay (jitter is bounded)

Stream

- Definition: A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission
- Common stream characteristics
  - Streams are unidirectional
  - A single source, and one or more sinks
  - Often, either the sink and/or source is a wrapper around hardware (e.g., CD device, TV monitor, dedicated storage)
- Two types of streams:
  - Simple: single flow of data, e.g., audio or video
  - Complex: multiple data flows, e.g., stereo audio or combination audio/video
Streams and QoS

- Streams are all about timely delivery of data

- Quality of Service (QoS) specifications
  - The required bit rate at which data should be transported.
  - The maximum delay until a session has been set up (i.e., when an application can start sending data).
  - The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
  - The maximum delay variance, or jitter.
  - The maximum round-trip delay.

Enforce QoS

- Network-level tools: prioritize packets (differentiated services)
- Use buffers to reduce jitter

Enforce QoS: Example

- Interleaving: multiple samples in a single packet: the effects of packet loss → spread the samples
Summary

## Fundamentals
- Client/Server communication protocols
  - Request vs. Request-reply vs. Request-reply-acknowledge
- Invocation semantics
  - Exact once vs. at least once vs. at most once
- Communication types
  - Transient vs. persistent
  - Synchronous vs. asynchronous

## Communication models
- RPC: remote procedure call
- Message-oriented communication
- Stream-Oriented communication
- Multicast communication