Distributed Systems
Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science
Room R4.20, steen@cs.vu.nl

Chapter 04: Communication

Version: November 5, 2009

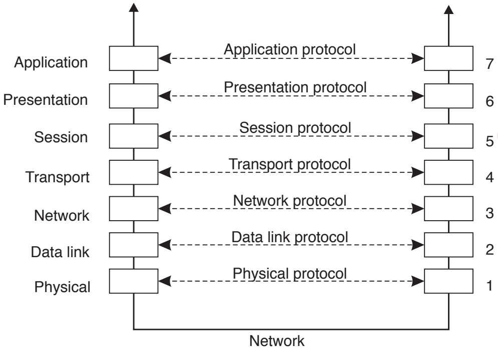
vrije Universiteit amsterdam

Contents

Chapter
01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems

Basic networking model

data translation
keep session alive
process link
routing
medium access
signal



e.g.,
on connection loss
recover connection,
close if not used for
long

Drawbacks

- Focus on message-passing only
- Often unneeded or unwanted functionality
- Violates access transparency

client needs IP addr and port of server

Low-level layers

Recap

- Physical layer: contains the specification and implementation of bits, and their transmission between sender and receiver
- Data link layer: prescribes the transmission of a series of bits into a frame to allow for error and flow control access to machines @ 1 hop
- Network layer: describes how packets in a network of computers are to be routed. access to machines @ n hop

Observation

For many distributed systems, the lowest-level interface is that of the network layer.

Transport Layer

Important

The transport layer provides the actual communication facilities for most distributed systems.

allows connection of remote processes

Standard Internet protocols

error detection, retransmission

guarantees delivery order

- TCP: connection-oriented, reliable, stream-oriented communication
- UDP: unreliable (best-effort) datagram communication

Note

IP multicasting is often considered a standard available service (which may be dangerous to assume).

special IP address X is associated with multiple IPs
message sent to X is automatically sent to those IPs

Middleware Layer

most often build on top of transport layer

Observation

Middleware is invented to provide **common** services and protocols that can be used by many different applications

- A rich set of communication protocols
- (Un)marshaling of data, necessary for integrated systems
- Naming protocols, to allow easy sharing of resources
- Security protocols for secure communication
- Scaling mechanisms, such as for replication and caching

each middleware implements its own

~ data translation

finding nodes
from names

Note

What remains are truly application-specific protocols...

Client/Server

Some observations

Client/Server computing is generally based on a model of transient synchronous communication:

- Client and server have to be active at time of commun. (transient)
- Client issues request and blocks until it receives reply
- Server essentially waits only for incoming requests, and subsequently processes them (synchronous)

Drawbacks 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)

Messaging

Message-oriented middleware

Aims at high-level persistent asynchronous communication: (persistent)

- Processes send each other messages, which are queued
- Sender need not wait for immediate reply, but can do other things
- Middleware often ensures fault tolerance

(asynchronous)

Remote Procedure Call (RPC)

- Basic RPC operation
- Parameter passing
- Variations

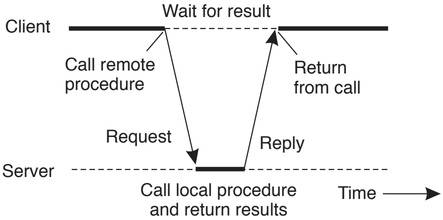
Basic RPC operation

Observations

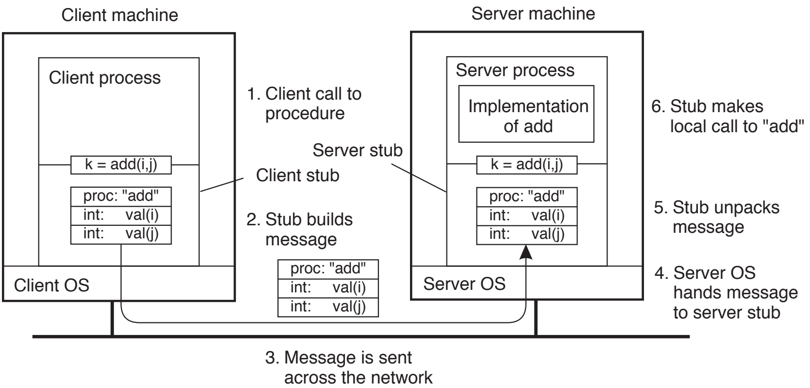
- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on separate machine

Conclusion

Communication between caller & callee can be hidden by using procedure-call mechanism.



Basic RPC operation



1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters and calls server.
6. Server returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client's OS.
9. Client's OS gives message to stub.
10. Client stub unpacks result and returns to the client.

RPC: Parameter passing

Parameter marshaling

There's more than just wrapping parameters into a message:

- Client and server machines may have different data representations (think of byte ordering)
- Wrapping a parameter means transforming a value into a sequence of bytes
- Client and server have to agree on the same encoding:
  - 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, transforming them into machine-dependent representations.

RPC: Parameter passing

RPC parameter passing: some assumptions

- Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
- All data that is to be operated on is passed by parameters. Excludes passing references to (global) data.

Conclusion

Full access transparency cannot be realized.

Observation

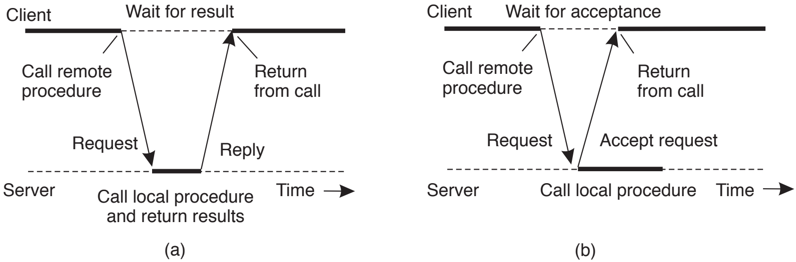
A remote reference mechanism enhances access transparency:

- Remote reference offers unified access to remote data
- Remote references can be passed as parameter in RPCs

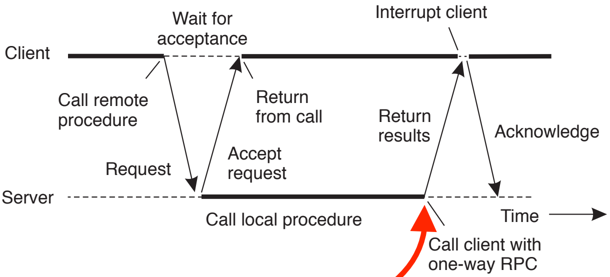
Asynchronous RPCs

Essence

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



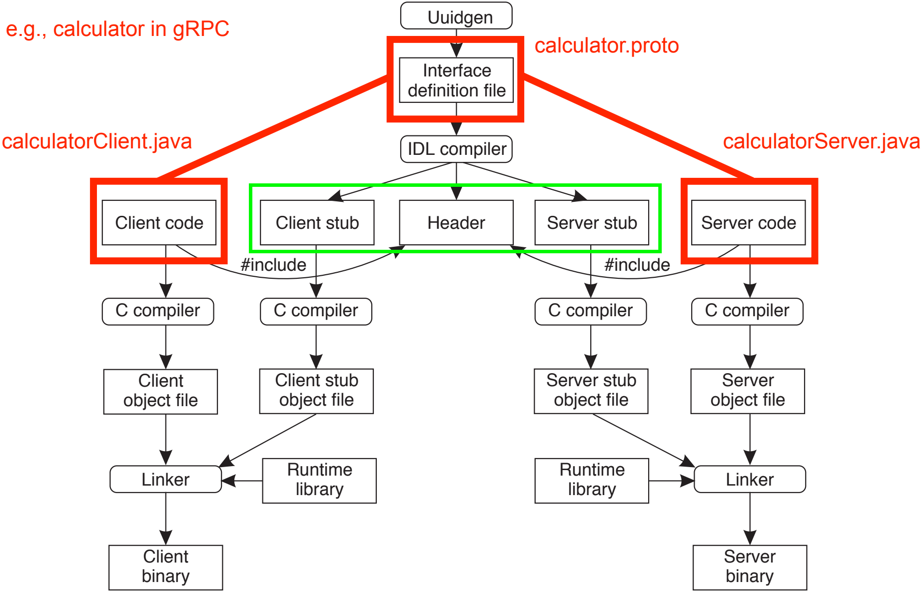
Deferred synchronous RPCs



Variation

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

RPC in practice



Message-Oriented Communication

- Transient Messaging
- Message-Queuing System
- Message Brokers
- Example: IBM Websphere

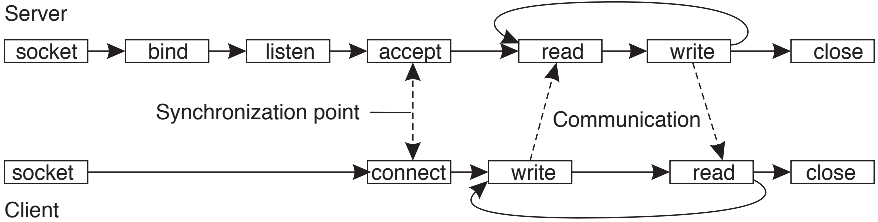
Transient messaging: sockets

Berkeley socket interface

SOCKET	Create a new communication endpoint
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept *N* connections
ACCEPT	Block until request to establish a connection
CONNECT	Attempt to establish a connection
SEND	Send data over a connection
RECEIVE	Receive data over a connection
CLOSE	Release the connection

Transient messaging: sockets

ServerSocket (Java)



Message-oriented middleware

Essence

Asynchronous persistent communication through support of middleware-level queues. Queues correspond to buffers at communication servers.

PUT	Append a message to a specified queue
GET	Block until the specified queue is nonempty, and remove the first message
POLL	Check a specified queue for messages, and remove the first. Never block
NOTIFY	Install a handler to be called when a message is put into the specified queue

Message broker

Observation

Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)

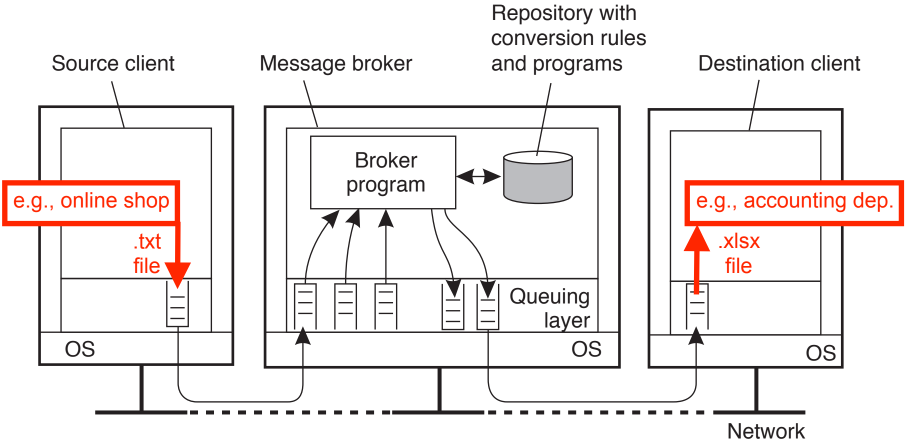
Message broker

Centralized component that takes care of application heterogeneity in an MQ system:

- Transforms incoming messages to target format
- Very often acts as an application gateway
- May provide subject-based routing capabilities ⇒ Enterprise Application Integration

Message broker

e.g., RabbitMQ



Stream-oriented communication

- Support for continuous media
- Streams in distributed systems
- Stream management

Continuous media

Observation

All communication facilities discussed so far are essentially based on a discrete, that is time-independent exchange of information

Continuous media

Characterized by the fact that values are time dependent:

- Audio
- Video
- Animations
- Sensor data (temperature, pressure, etc.)

Continuous media

Transmission modes

Different timing guarantees with respect to data transfer:

- 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.

Some common stream characteristics

- Streams are unidirectional
- There is generally a single source, and one or more sinks
- Often, either the sink and/or source is a wrapper around hardware (e.g., camera, CD device, TV monitor)
- Simple stream: a single flow of data, e.g., audio or video
- Complex stream: multiple data flows, e.g., stereo audio or combination audio/video

Streams and QoS

Essence

Streams are all about timely delivery of data. How do you specify this Quality of Service (QoS)? Basics:

- 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.

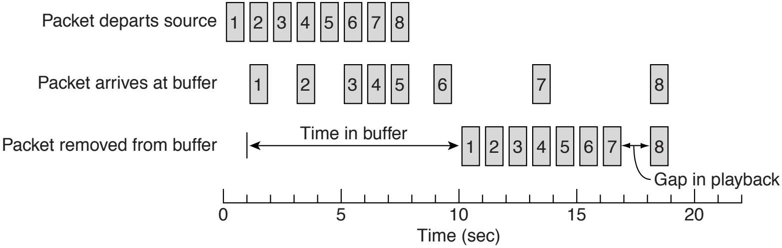
Enforcing QoS

Observation

There are various network-level tools, such as differentiated services by which certain packets can be prioritized.

Also

Use buffers to reduce jitter:

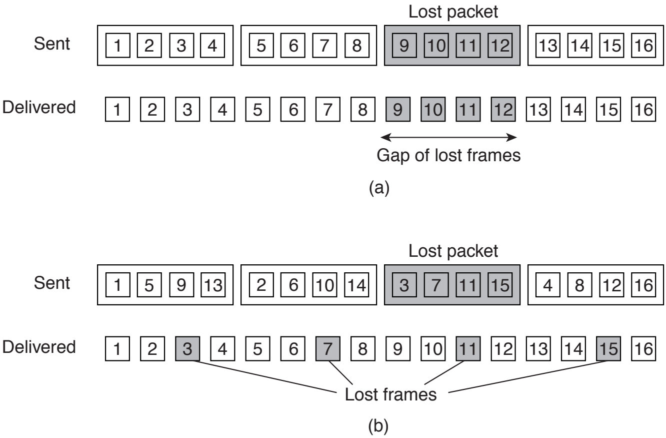


Enforcing QoS

Problem

How to reduce the effects of packet loss (when multiple samples are in a single packet)?

Enforcing QoS



Stream synchronization

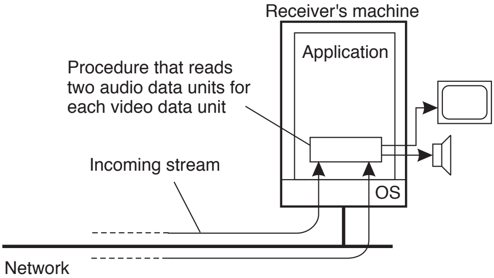
Problem

Given a complex stream, how do you keep the different substreams in synch?

Example

Think of playing out two channels, that together form stereo sound. Difference should be less than 20–30 μsec!

Stream synchronization



Alternative

Multiplex all substreams into a single stream, and demultiplex at the receiver. Synchronization is handled at multiplexing/demultiplexing point (MPEG).

Epidemic Algorithms

- General background
- Update models
- Removing objects

Principles

Basic idea

Assume there are no write–write conflicts:

- Update operations are performed at a single server
- A replica passes updated state to only a few neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every replica

Two forms of epidemics

- Anti-entropy: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards
- Gossiping: A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).

Anti-entropy

Principle operations

- A node P selects another node Q from the system at random.
- Push: P only sends its updates to Q
- Pull: P only retrieves updates from Q
- Push-Pull: P and Q exchange mutual updates (after which they hold the same information).

Observation

For push-pull it takes $\mathcal{O}(log(N))$ rounds to disseminate updates to all N nodes (round = when every node has taken the initiative to start an exchange).

Gossiping

Basic model

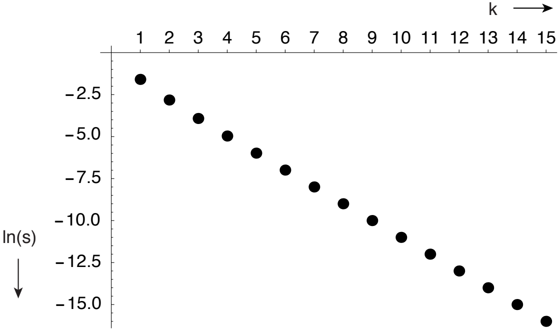
A server S having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, S stops contacting other servers with probability $1/k$.

Observation

If s is the fraction of ignorant servers (i.e., which are unaware of the update), it can be shown that with many servers

$$s = e^{-(k+1)(1-s)}$$

Gossiping



Consider 10,000 nodes		
k	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

Note

If we really have to ensure that all servers are eventually updated, gossiping alone is not enough