Distributed Systems
Principles and Paradigms

Maarten van Steen

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

Chapter 04: Communication

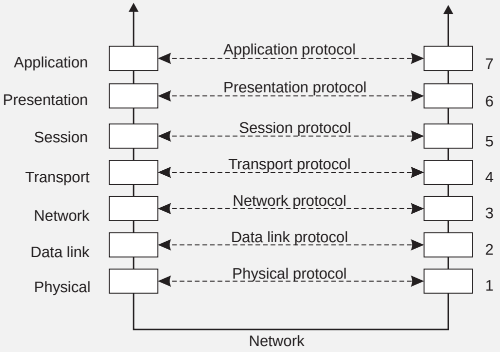
Version: November 5, 2012

vrije Universiteit amsterdam

Layered Protocols

- Low-level layers
- Transport layer
- Application layer
- Middleware layer

Basic networking model



Drawbacks

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

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
- Network layer: describes how packets in a network of computers are to be routed.

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.

Standard Internet protocols

- 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).

Middleware Layer

Observation

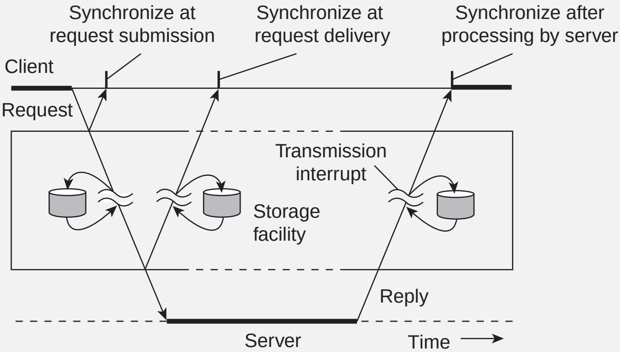
Middleware is invented to provide common services and protocols that can be used by many different applications

- 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

Note

What remains are truly application-specific protocols...
such as?

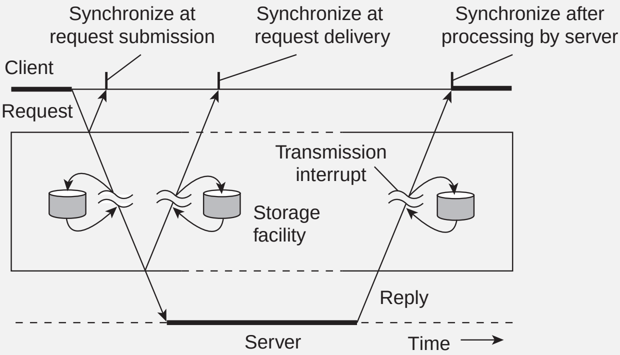
Types of communication



Distinguish

- Transient versus persistent communication
- Asynchrounous versus synchronous communication

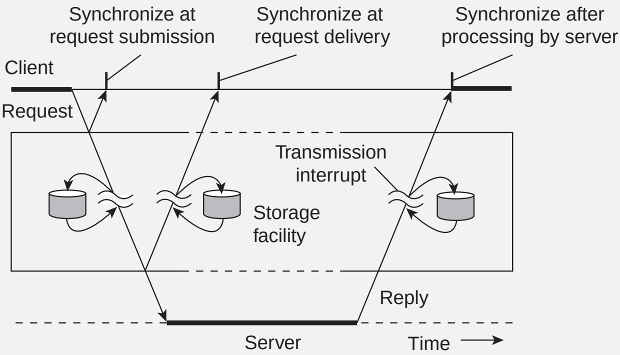
Types of communication



Transient versus persistent

- Transient communication: Comm. server discards message when it cannot be delivered at the next server, or at the receiver.
- Persistent communication: A message is stored at a communication server as long as it takes to deliver it.

Types of communication



Places for synchronization

- At request submission
- At request delivery
- After request processing

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.
- Client issues request and blocks until it receives reply
- Server essentially waits only for incoming requests, and subsequently processes them

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)

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.
- Client issues request and blocks until it receives reply
- Server essentially waits only for incoming requests, and subsequently processes them

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:

- 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

Remote Procedure Call (RPC)

- Basic RPC operation
- Parameter passing
- Variations

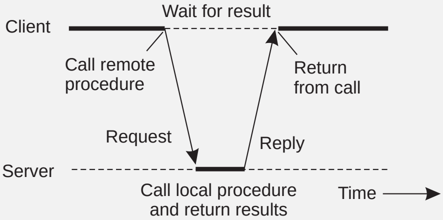
Basic RPC operation

Observations

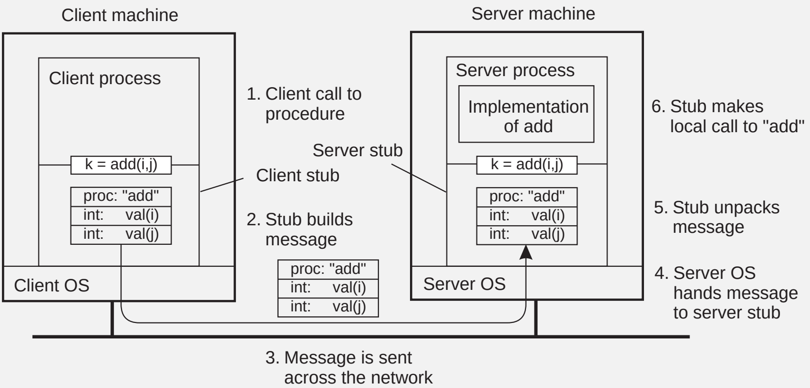
- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on separate machine

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 makes local call and 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

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

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

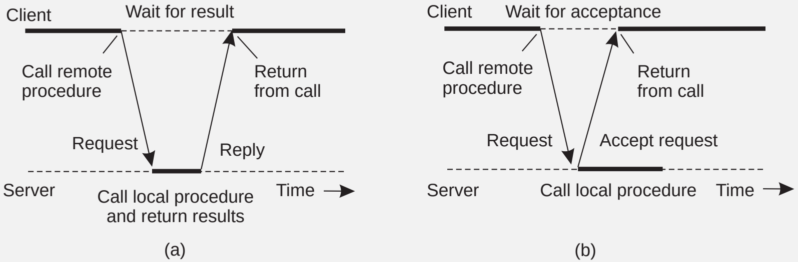
A remote reference mechanism enhances access transparency:

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

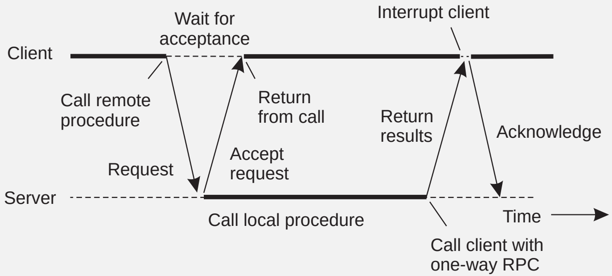
Asynchronous RPCs

Essence

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



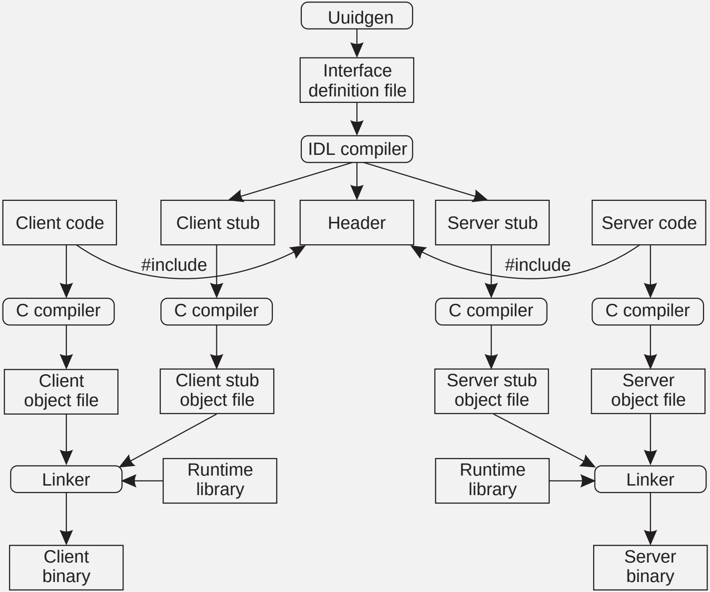
Deferred synchronous RPCs



Variation

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

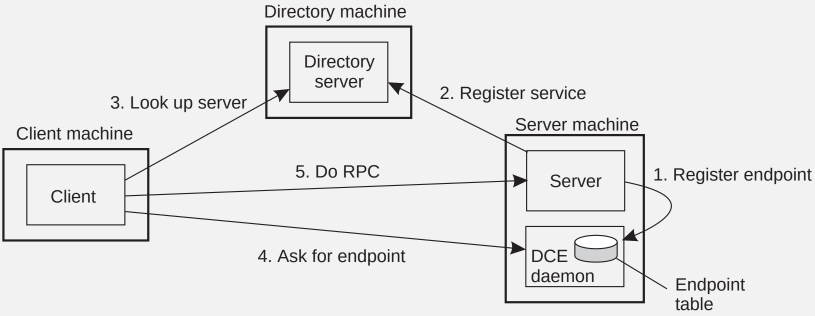
RPC in practice



Client-to-server binding (DCE)

Issues

(1) Client must locate server machine, and (2) locate the server.



Message-Oriented Communication

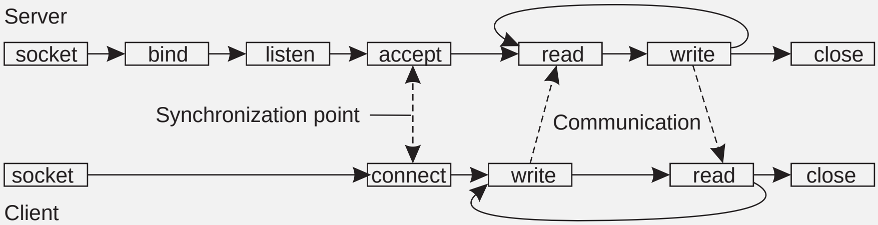
- Transient Messaging
- Message-Queuing System
- Message Brokers
- Example: IBM Websphere

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



Sockets: Python code

Server

```
import socket
HOST = ''
PORT = SERVERPORT
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(N)                # listen to max N queued connection
(conn, addr) = s.accept() # returns new socket + addr client
while 1: # forever
  data = conn.recv(1024)
  if not data: break
  conn.send(data)
conn.close()
```

Client

```
import socket
HOST = 'distsys.cs.vu.nl'
PORT = SERVERPORT
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
```

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

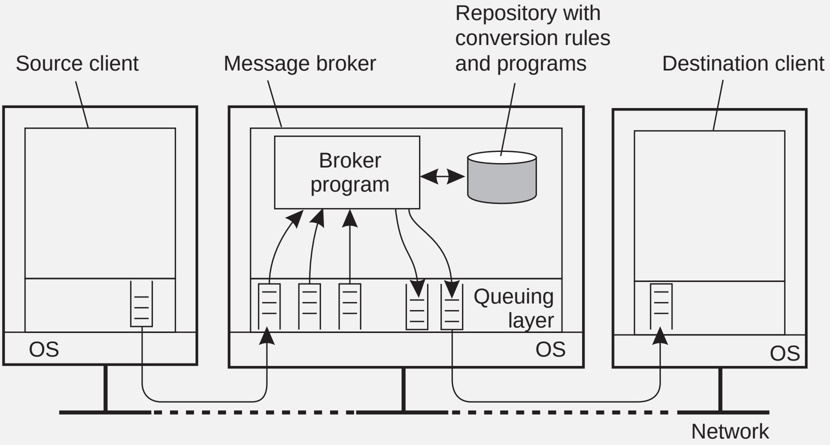
Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)

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



IBM's WebSphere MQ

Basic concepts

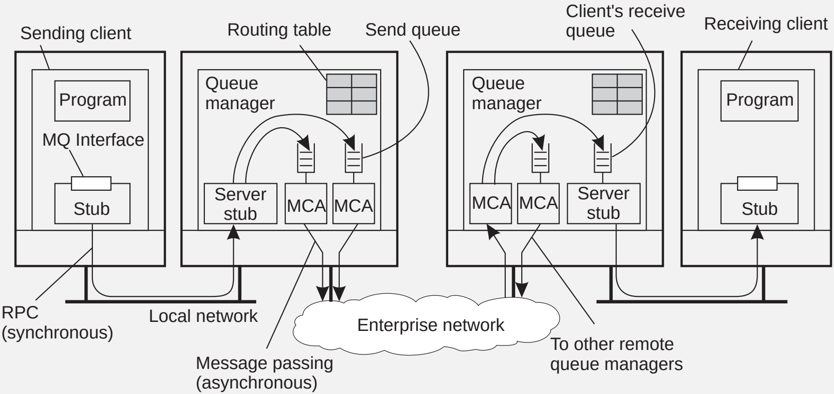
- Application-specific messages are put into, and removed from queues
- Queues reside under the regime of a queue manager
- Processes can put messages only in local queues, or through an RPC mechanism

IBM's WebSphere MQ

Message transfer

- Messages are transferred between queues
- Message transfer between queues at different processes, requires a channel
- At each endpoint of channel is a message channel agent
- Message channel agents are responsible for:
  - Setting up channels using lower-level network communication facilities (e.g., TCP/IP)
  - (Un)wrapping messages from/in transport-level packets
  - Sending/receiving packets

IBM's WebSphere MQ

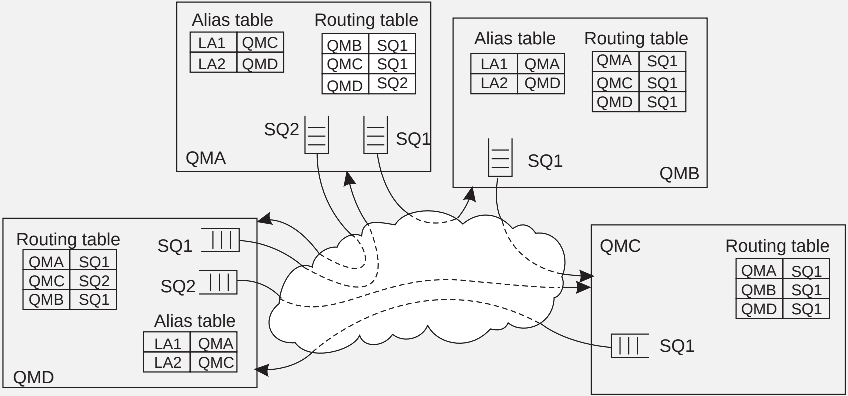


- Channels are inherently unidirectional
- Automatically start MCAs when messages arrive
- Any network of queue managers can be created
- Routes are set up manually (system administration)

IBM's WebSphere MQ

Routing

By using logical names, in combination with name resolution to local queues, it is possible to put a message in a remote queue



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.

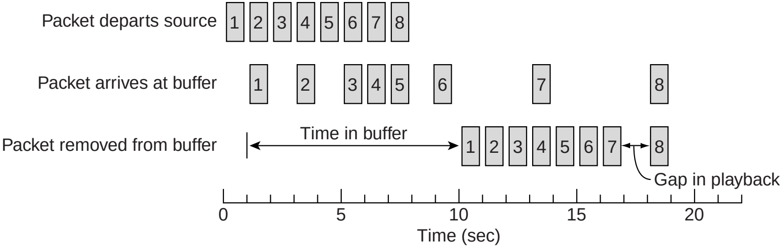
Enforcing QoS

Observation

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

Also

Use buffers to reduce jitter:

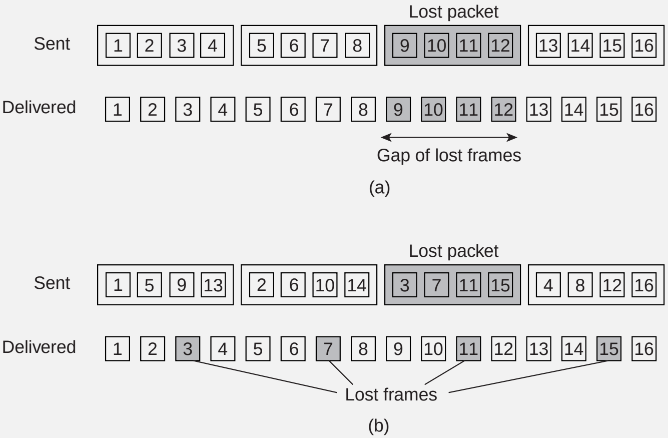


Enforcing QoS

Problem

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

Enforcing QoS



Stream synchronization

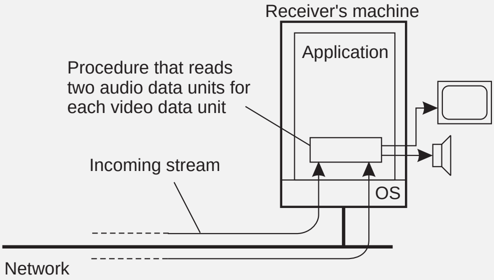
Problem

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!

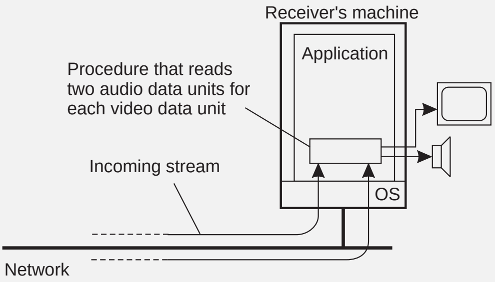
Stream synchronization



Alternative

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

Stream synchronization



Alternative

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

Multicast communication

- Application-level multicasting
- Gossip-based data dissemination

Application-level multicasting

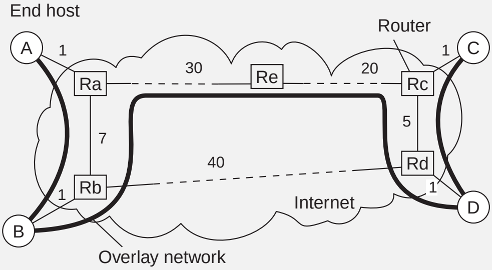
Essence

Organize nodes of a distributed system into an overlay network and use that network to disseminate data.

Chord-based tree building

1. Initiator generates a multicast identifier *mid*.
2. Lookup *succ(mid)*, the node responsible for *mid*.
3. Request is routed to *succ(mid)*, which will become the root.
4. If *P* wants to join, it sends a join request to the root.
5. When request arrives at *Q*:
   - *Q* has not seen a join request before $\Rightarrow$ it becomes forwarder; *P* becomes child of *Q*. Join request continues to be forwarded.
   - *Q* knows about tree $\Rightarrow$ *P* becomes child of *Q*. No need to forward join request anymore.

ALM: Some costs



- Link stress: How often does an ALM message cross the same physical link? Example: message from *A* to *D* needs to cross $\langle Ra, Rb \rangle$ twice.
- Stretch: Ratio in delay between ALM-level path and network-level path. Example: messages *B* to *C* follow path of length 71 at ALM, but 47 at network level $\Rightarrow$ stretch = 71/47.

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 as taken the initiative to start an exchange).

Anti-entropy: analysis (extra)

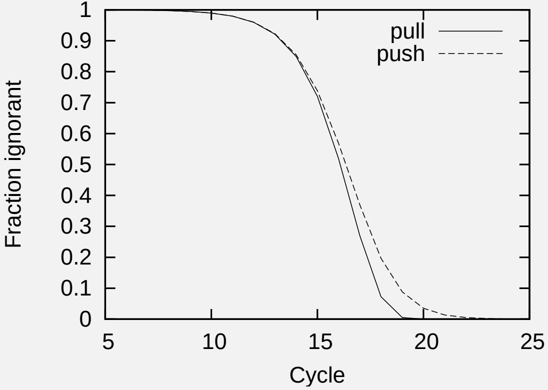
Basics

Consider a single source, propagating its update. Let p_i be the probability that a node has not received the update after the i-th cycle.

Analysis: staying ignorant

- With pull, $p_{i+1} = (p_i)^2$: the node was not updated during the i-th cycle and should contact another ignorant node during the next cycle.
- With push, $p_{i+1} = p_i(1 - \frac{1}{N})^{N(1-p_i)} \approx p_i e^{-1}$ (for small p_i and large N): the node was ignorant during the i-th cycle and no updated node chooses to contact it during the next cycle.
- With push-pull: $(p_i)^2 \cdot (p_i e^{-1})$

Anti-entropy performance



Gossiping

Basic model

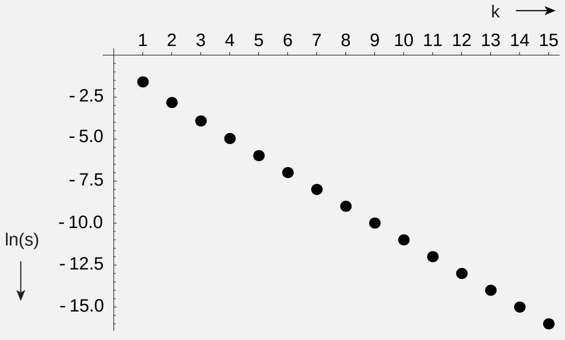
A server S having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, S stops contacting other servers with probability $1/k$.

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

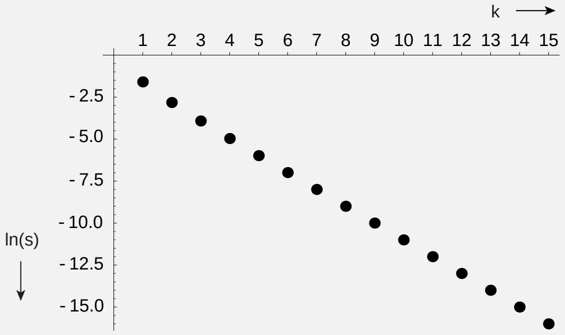


Consider 10,000 nodes		
k	*s*	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

Note

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

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

Deleting values

Fundamental problem

We cannot remove an old value from a server and expect the removal to propagate. Instead, mere removal will be undone in due time using epidemic algorithms

Solution

Removal has to be registered as a special update by inserting a death certificate

Deleting values

Next problem

When to remove a death certificate (it is not allowed to stay for ever):

- Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)
- Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers)

Note

It is necessary that a removal actually reaches all servers.

Question

What's the scalability problem here?

Example applications

Typical apps

- Data dissemination: Perhaps the most important one. Note that there are many variants of dissemination.
- Aggregation: Let every node i maintain a variable x_i. When two nodes gossip, they each reset their variable to

$$x_i, x_j \leftarrow (x_i + x_j)/2$$

Result: in the end each node will have computed the average $\bar{x} = \sum_i x_i / N$.

Example application: aggregation

Aggregation (continued)

When two nodes gossip, they each reset their variable to

$$x_i, x_j \leftarrow (x_i + x_j)/2$$

Result: in the end each node will have computed the average $\bar{x} = \sum_i x_i / N$.

Question

What happens if initially $x_i = 1$ and $x_j = 0, j \neq i$?

Question

How can we start this computation without pre-assigning a node i to start as only one with $x_i \leftarrow 1$?

Example application: aggregation

Aggregation (continued)

When two nodes gossip, they each reset their variable to

$$x_i, x_j \leftarrow (x_i + x_j)/2$$

Result: in the end each node will have computed the average $\bar{x} = \sum_i x_i / N$.

Question

What happens if initially $x_i = 1$ and $x_j = 0, j \neq i$?

Question

How can we start this computation without pre-assigning a node i to start as only one with $x_i \leftarrow 1$?

Example application: aggregation

Aggregation (continued)

When two nodes gossip, they each reset their variable to

$$x_i, x_j \leftarrow (x_i + x_j)/2$$

Result: in the end each node will have computed the average $\bar{x} = \sum_i x_i / N$.

Question

What happens if initially $x_i = 1$ and $x_j = 0, j \neq i$?

Question

How can we start this computation without pre-assigning a node i to start as only one with $x_i \leftarrow 1$?