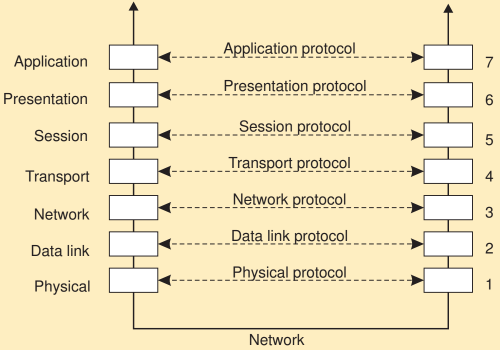
# Distributed Systems

(3rd Edition)

## Chapter 04: Communication

Version: August 29, 2017

# 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

# Middleware layer

## Observation

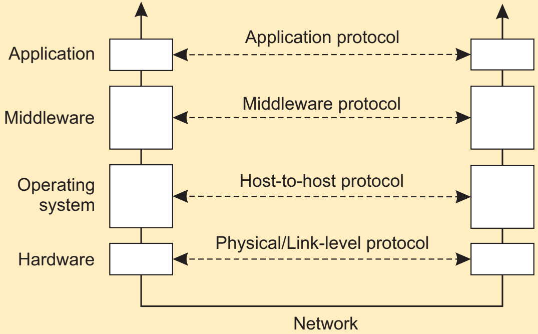
Middleware is invented to provide common services and protocols that can be used by many different applications

- 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

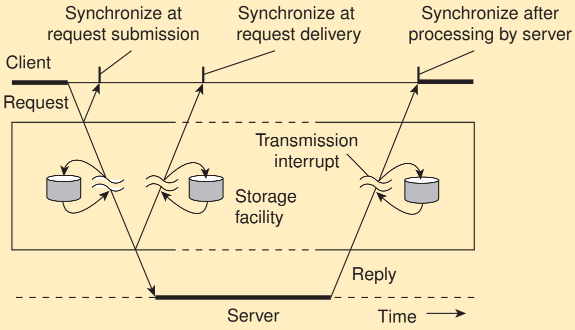
What remains are truly application-specific protocols... such as?

# An adapted layering scheme



# Types of communication

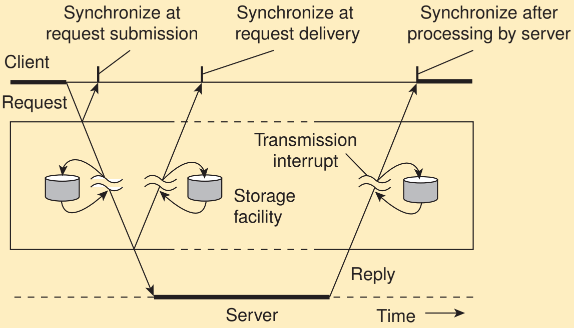
## Distinguish...




- Transient versus persistent communication
- Asynchronous versus synchronous communication

# Types of communication

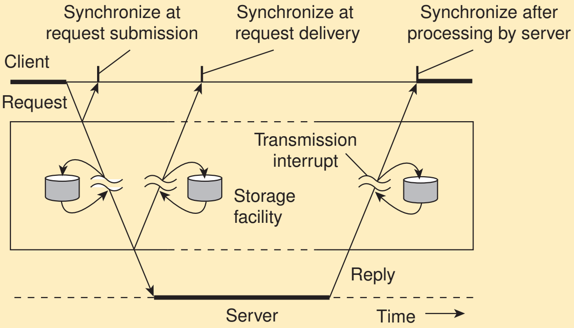
## Transient versus persistent




- 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 communication
- 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

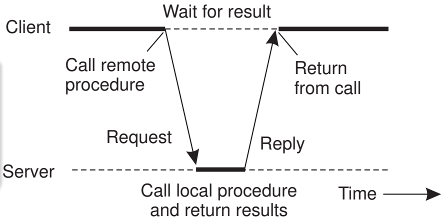
# Basic RPC operation

## Observations

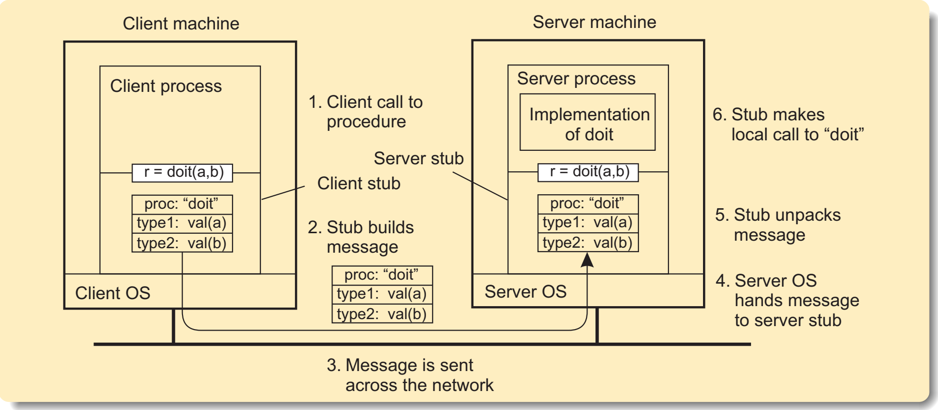
- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on separate machine

## 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; calls server.
6. Server does local call; 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; returns to client.

# RPC: Parameter passing

## 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)

## Conclusion

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

# 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.

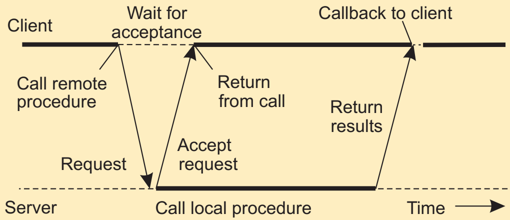
### A remote reference mechanism enhances access transparency

- Remote reference offers unified access to remote data
- Remote references can be passed as parameter in RPCs
- Note: stubs can sometimes be used as such references

# Asynchronous RPCs

## Essence

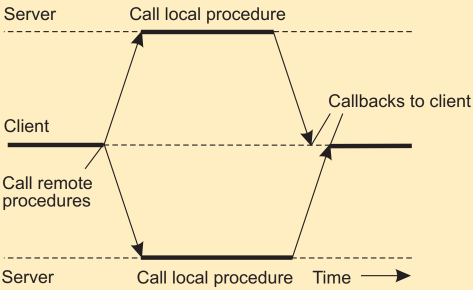
Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.



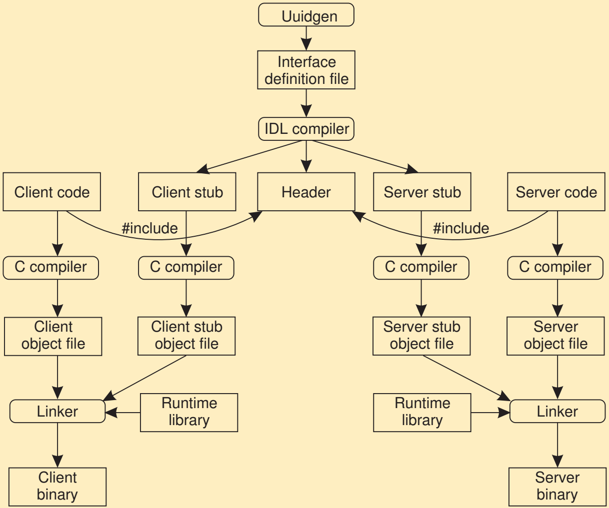
# Sending out multiple RPCs

## Essence

Sending an RPC request to a group of servers.



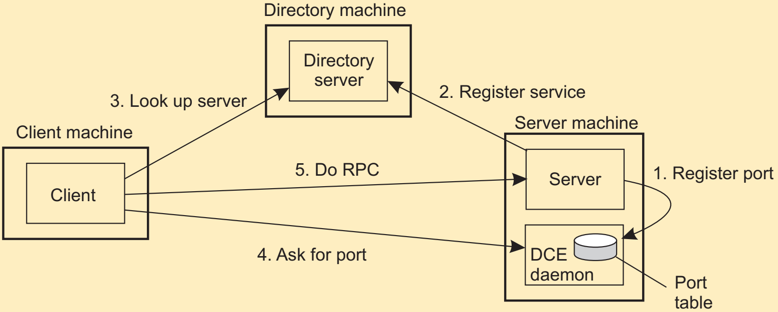
# RPC in practice



# Client-to-server binding (DCE)

## Issues

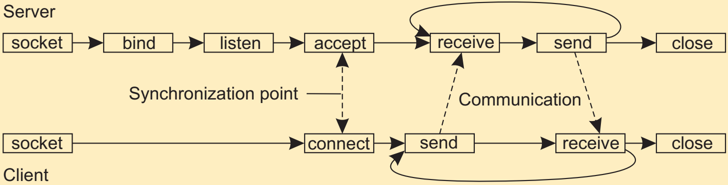
(1) Client must locate server machine, and (2) locate the server.



# Transient messaging: sockets

## Berkeley socket interface

| Operation | Description |
|---|---|
| `socket` | Create a new communication end point |
| `bind` | Attach a local address to a socket |
| `listen` | Tell operating system what the maximum number of pending connection requests should be |
| `accept` | Block caller until a connection request arrives |
| `connect` | Actively attempt to establish a connection |
| `send` | Send some data over the connection |
| `receive` | Receive some data over the connection |
| `close` | Release the connection |



# Sockets: Python code

## Server

```
from socket  import *
s = socket(AF_INET, SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
(conn, addr) = s.accept()  # returns new socket and addr. client
while True:                # forever
  data = conn.recv(1024)   # receive data from client
  if not data: break       # stop if client stopped
  conn.send(str(data)+"*") # return sent data plus an "*"
conn.close()               # close the connection
```

## Client

```
from socket  import *
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT)) # connect to server (block until accepted)
s.send('Hello, world')  # send same data
data = s.recv(1024)     # receive the response
print data              # print the result
s.close()               # close the connection
```

# Making sockets easier to work with

## Observation

Sockets are rather low level and programming mistakes are easily made. However, the way that they are used is often the same (such as in a client-server setting).

## Alternative: ZeroMQ

Provides a higher level of expression by pairing sockets: one for sending messages at process $P$ and a corresponding one at process $Q$ for receiving messages. All communication is asynchronous.

## Three patterns

- Request-reply
- Publish-subscribe
- Pipeline

# Request-reply

## Server

```
import zmq
context = zmq.Context()

p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
p2 = "tcp://"+ HOST +":"+ PORT2 # how and where to connect
s  = context.socket(zmq.REP)    # create reply socket

s.bind(p1)                      # bind socket to address
s.bind(p2)                      # bind socket to address
while True:
  message = s.recv()            # wait for incoming message
  if not "STOP" in message:     # if not to stop...
    s.send(message + "*")       # append "*" to message
  else:                         # else...
    break                       # break out of loop and end
```

# Request-reply

## Client

```python
import zmq
context = zmq.Context()

php = "tcp://"+ HOST +":"+ PORT # how and where to connect
s   = context.socket(zmq.REQ)    # create socket

s.connect(php)                   # block until connected
s.send("Hello World")            # send message
message = s.recv()               # block until response
s.send("STOP")                   # tell server to stop
print message                    # print result
```

# Publish-subscribe

## Server

```python
import zmq, time

context = zmq.Context()
s = context.socket(zmq.PUB)         # create a publisher socket
p = "tcp://"+ HOST +":"+ PORT       # how and where to communicate
s.bind(p)                           # bind socket to the address
while True:
  time.sleep(5)                     # wait every 5 seconds
  s.send("TIME " + time.asctime()) # publish the current time
```

## Client

```python
import zmq

context = zmq.Context()
s = context.socket(zmq.SUB)             # create a subscriber socket
p = "tcp://"+ HOST +":"+ PORT           # how and where to communicate
s.connect(p)                            # connect to the server
s.setsockopt(zmq.SUBSCRIBE, "TIME")    # subscribe to TIME messages

for i in range(5):  # Five iterations
  time = s.recv()   # receive a message
  print time
```

# Pipeline

## Source

```python
import zmq, time, pickle, sys, random

context = zmq.Context()
me  = str(sys.argv[1])
s   = context.socket(zmq.PUSH)        # create a push socket
src = SRC1  if me == '1' else SRC2    # check task source host
prt = PORT1 if me == '1' else PORT2   # check task source port
p   = "tcp://"+ src +":"+ prt         # how and where to connect
s.bind(p)                             # bind socket to address

for i in range(100):                  # generate 100 workloads
  workload = random.randint(1, 100)   # compute workload
  s.send(pickle.dumps((me,workload))) # send workload to worker
```

# Pipeline

## Worker

```python
import zmq, time, pickle, sys

context = zmq.Context()
me = str(sys.argv[1])
r  = context.socket(zmq.PULL)      # create a pull socket
p1 = "tcp://"+ SRC1 +":"+ PORT1    # address first task source
p2 = "tcp://"+ SRC2 +":"+ PORT2    # address second task source
r.connect(p1)                      # connect to task source 1
r.connect(p2)                      # connect to task source 2

while True:
  work = pickle.loads(r.recv())    # receive work from a source
  time.sleep(work[1]*0.01)         # pretend to work
```

# MPI: When lots of flexibility is needed

## Representative operations

| Operation | Description |
|---|---|
| MPI_bsend | Append outgoing message to a local send buffer |
| MPI_send | Send a message and wait until copied to local or remote buffer |
| MPI_ssend | Send a message and wait until transmission starts |
| MPI_sendrecv | Send a message and wait for reply |
| MPI_isend | Pass reference to outgoing message, and continue |
| MPI_issend | Pass reference to outgoing message, and wait until receipt starts |
| MPI_recv | Receive a message; block if there is none |
| MPI_irecv | Check if there is an incoming message, but do not block |

# Message-oriented middleware

## Essence

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

## Operations

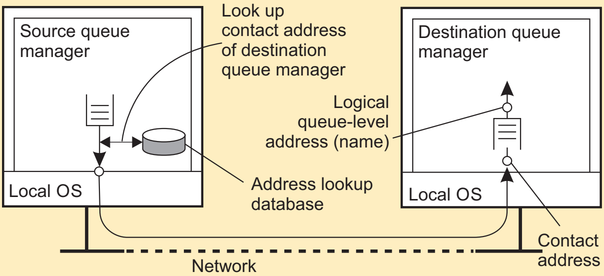
| Operation | Description |
| --- | --- |
| `put` | Append a message to a specified queue |
| `get` | Block until the specified queue is nonempty, and remove the first message |
| `poll` | Check a specified queue for messages, and remove the first. Never block |
| `notify` | Install a handler to be called when a message is put into the specified queue |

# General model

## Queue managers

Queues are managed by queue managers. An application can put messages only into a local queue. Getting a message is possible by extracting it from a local queue only ⇒ queue managers need to route messages.

## Routing



# Message broker

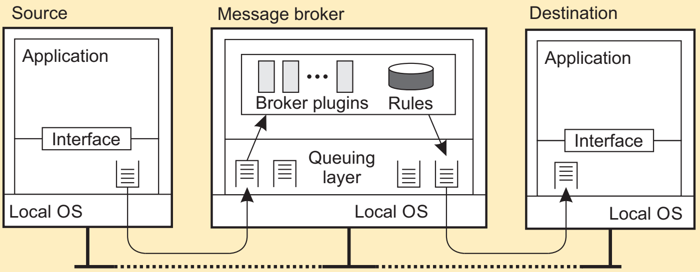
### Observation

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

### Broker handles 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 (i.e., publish-subscribe capabilities)

# Message broker: general architecture



# IBM's WebSphere MQ

## Basic concepts

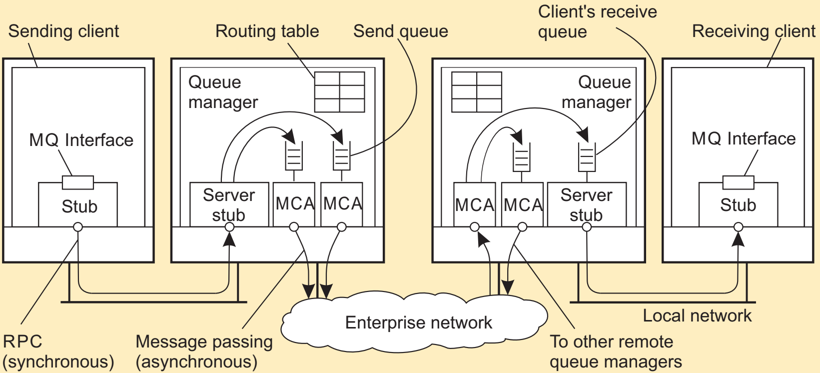
- Application-specific messages are put into, and removed from queues
- Queues reside under the regime of a queue manager
- Processes can put messages only in local queues, or through an RPC mechanism

## Message transfer

- Messages are transferred between queues
- Message transfer between queues at different processes, requires a channel
- At each end point 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

## Schematic overview




- 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)

# Message channel agents

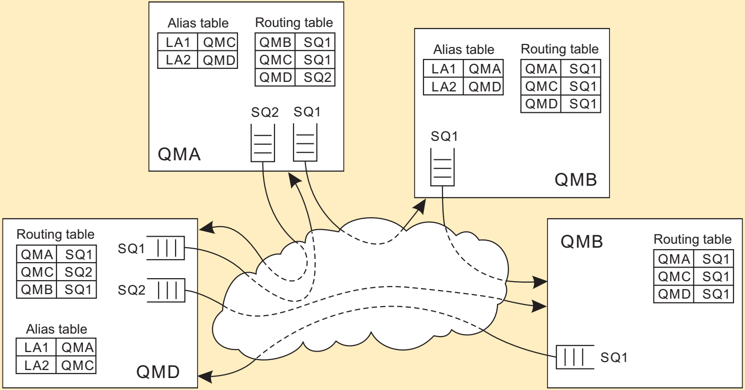
## Some attributes associated with message channel agents

| Attribute | Description |
|---|---|
| Transport type | Determines the transport protocol to be used |
| FIFO delivery | Indicates that messages are to be delivered in the order they are sent |
| Message length | Maximum length of a single message |
| Setup retry count | Specifies maximum number of retries to start up the remote MCA |
| Delivery retries | Maximum times MCA will try to put received message into queue |

# 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



# Application-level multicasting

## Essence

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

- Oftentimes a tree, leading to unique paths
- Alternatively, also mesh networks, requiring a form of routing

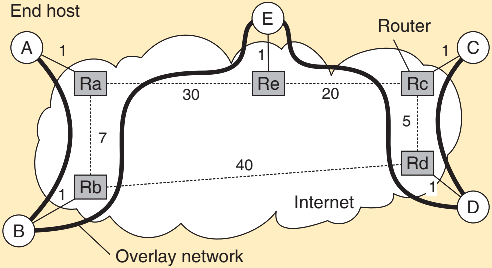
# Application-level multicasting in Chord

## Basic approach

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 ⇒ it becomes forwarder; *P* becomes child of *Q*. Join request continues to be forwarded.
   - *Q* knows about tree ⇒ *P* becomes child of *Q*. No need to forward join request anymore.

# ALM: Some costs

## Different metrics




- 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 73 at ALM, but 47 at network level $\Rightarrow$ stretch = 73/47.

# Flooding

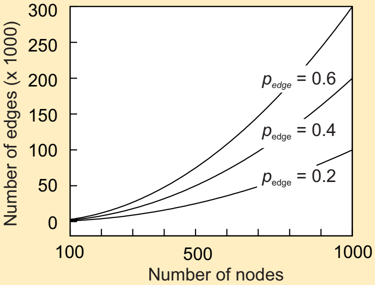
## Essence

$P$ simply sends a message $m$ to each of its neighbors. Each neighbor will forward that message, except to $P$, and only if it had not seen $m$ before.

## Performance

The more edges, the more expensive!

## The size of a random overlay as function of the number of nodes




## Variation

Let $Q$ forward a message with a certain probability $p_{flood}$, possibly even dependent on its own number of neighbors (i.e., node degree) or the degree of its neighbors.

# Epidemic protocols

## 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
- Rumor spreading: 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.
- Pull: $P$ only pulls in new updates from $Q$
- Push: $P$ only pushes its own updates to $Q$
- Push-pull: $P$ and $Q$ send updates to each other

## 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).

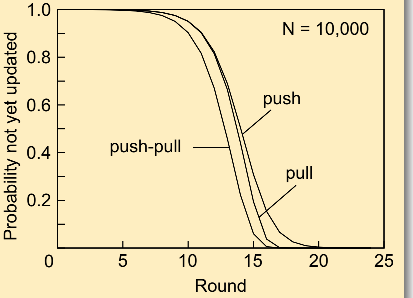
# Anti-entropy: analysis

## Basics

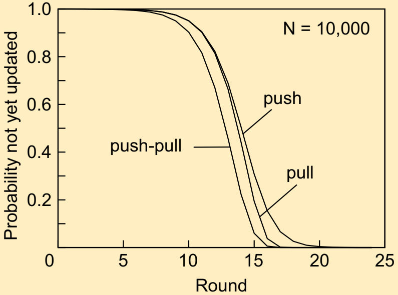
Consider a single source, propagating its update. Let $p_i$ be the probability that a node has not received the update after the $i^{th}$ round.

## Analysis: staying ignorant

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



# Anti-entropy performance



# Rumor spreading

## 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 $p_{stop}$.

## 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

$$s = e^{-(1/p_{stop}+1)(1-s)}$$

# Formal analysis

## Notations

Let $s$ denote fraction of nodes that have not yet been updated (i.e., susceptible; $i$ the fraction of updated (infected) and active nodes; and $r$ the fraction of updated nodes that gave up (removed).

## From theory of epidemics

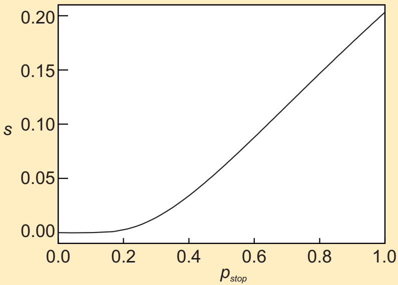
$$
\begin{array}{llcl}
(1) & ds/dt & = & -s \cdot i \\
(2) & di/dt & = & s \cdot i - p_{stop} \cdot (1-s) \cdot i \\
\Rightarrow & di/ds & = & -(1+p_{stop}) + \frac{p_{stop}}{s} \\
\Rightarrow & i(s) & = & -(1+p_{stop}) \cdot s + p_{stop} \cdot \ln(s) + C
\end{array}
$$

## Wrapup

$i(1) = 0 \Rightarrow C = 1 + p_{stop} \Rightarrow i(s) = (1+p_{stop}) \cdot (1-s) + p_{stop} \cdot \ln(s)$. We are looking for the case $i(s) = 0$, which leads to $s = e^{-(1/p_{stop}+1)(1-s)}$

# Rumor spreading

## The effect of stopping




| Consider 10,000 nodes | | |
|---|---|---|
| $1/p_{stop}$ | $s$ | $N_s$ |
| 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, rumor spreading 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

## 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.