Introduction to Distributed Systems

February 9, 2021

Summary

- What is a distributed system?
- Examples
- Why distribution?
- Challenges
- Further Reading

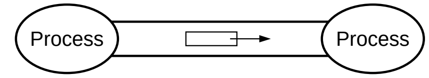
Summary

- What is a distributed system?
- Examples
- Why distribution?
- Challenges
- Further Reading

Distributed System

Definition A distributed system consists of a **collection of** distinct **processes** which are spatially separated and **which communicate with one another by exchanging messages**. (L. Lamport, "Time, Clocks and the Order of Events in a Distributed System", CACM)

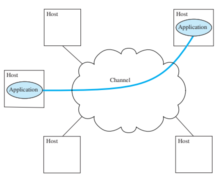
- "A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process."



Message based communication

Message a sequence of bits

- Whose format and meaning are specified by a *communication a protocol*
- That is transported from its source to its destination by a **communications network**



Summary

- What is a distributed system?
- Examples
- Why distribution?
- Challenges
- Further Reading

Examples?

Other (?) Distributed Systems/Applications.

- Web and Internet
- Google's search service
- Google's voice-to-text service
- *Email service*
- *Peer-to-peer* applications, such as Bittorrent
- FEUP's file system
- Telecommunication networks
- ATM networks (SIBS)
- Home automation (IoT)
- Factory automation (Industry 4.0)
- Fly-by-wire, drive-by-wire. (Autonomous driving)

Summary

- What is a distributed system?
- Examples
- Why distribution?
- Challenges
- Further Reading

Potencial Advantages

- Sharing of resources
- Access to remote resources
- Performance
  - Can use multiple computers to solve a problem
- Scalability:
  - Load (no. of users/request rate)
  - Geographical;
  - Administrative
- Fault tolerance
  - Reliability
  - Availability

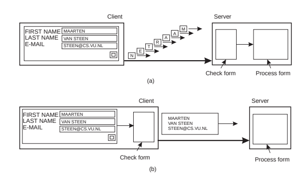
Scalability: Challenges

- Centralization
  - processing;
  - data;
  - algorithms.
- Synchronous comunication
- Security and (lack of) trust

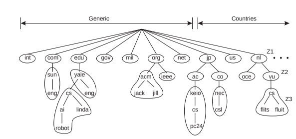
Scalability: Some Techniques (1/2)

Distribuition

processing:



data (partitioning):



Scalability: Some Techniques (2/2)

- Distributed (decentralized) algorithms:
  - System global state is unknown (relativity)
    - Can use only information locally available
  - Correctness must be ensured even in the presence of faults
  - No single physical clock
- Asynchronous communication
- Replication and *caches*:
  - \+ reduces communication latency;
  - \+ allow distributed processing;
  - \- raises consistency problemss

Summary

- What is a distributed system?
- Examples
- Why distribution?
- Challenges
- Further Reading

Challenges

- Partial failures
  - Some components may fail, while others continue to operate correctly
- IPC latency
  - IPC across the network has a larger and unpredictable latency, which usually cannot be bounded
- No global time
- No shared physical memory and distinct address spaces
  - Pointers are meaningful only in the context of the respective address space
- Heterogeneity
  - Has several facets
- Lack of security and trust

Summary

- What is a distributed system?
- Examples
- Why distribution?
- Challenges
- Further Reading

Further Reading

- *Distributed Systems, 3rd Ed.*, Chapter 1
- Michael Schroeder (et. al.) State-of-the-Art Distributed System: Computing with BOB
  - Nice "vision" from leading distributed system's researchers of DEC's SRC around 1990
  - Read only Sections 1 and 2
- Jim Waldo, et. al, A Note on Distributed Computing
  - Somewhat language-oriented, by people who designed Java RMI

Communication Channels

February 16, 2021

Roadmap

Message-based communication

Properties of a Communication Channel

Internet Protocols
UDP
TCP

Multimedia Applications and End-to-end Argument

Roadmap

Message-based communication

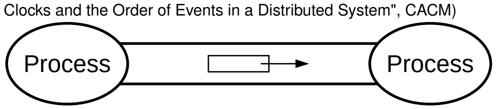
Properties of a Communication Channel

Internet Protocols
UDP
TCP

Multimedia Applications and End-to-end Argument

Distributed System

Definition A distributed system consists of a **collection of** distinct **processes** which are spatially separated and **which communicate with one another by exchanging messages**. (L. Lamport, "Time, Clocks and the Order of Events in a Distributed System", CACM)

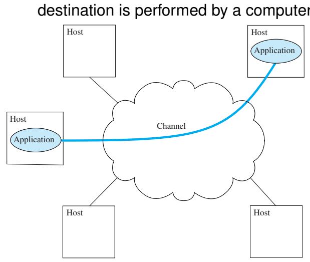


What is a message? is an **atomic** bit string

- Its format and its meaning are specified by a communications protocol

Message-based Communication and Networking

- The transport of a message from its source to its destination is performed by a computer network.



- The network can be abstracted as a communication channel
  - What are the properties of such a channel?

Distributed Computing Fallacies (Sun People)

Some assumptions proved wrong:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. The topology does not change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

- For more information, you can check:
  - Arnon Rotem-Gal-Oz, Fallacies of Distributed Computing Explained
    - Too much hype, almost ... "evangelist" style
  - Stephen Asbury, The Eight Fallacies of Distributed Computing
    - An entertaining talk.

Latency

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

src:Jeffrey Dean presentation at LADIS09

Roadmap

Message-based communication

Properties of a Communication Channel

Internet Protocols
UDP
TCP

Multimedia Applications and End-to-end Argument

(Some) Properties of a Communication Channel

- Connection-based vs. connectionless
- Reliable vs. unreliable (may loose messages)
- Ensures order (or not)
- Message-based vs. stream-based
- With or without flow control
- Number of ends of the channel

Question Why these properties?

Connection-based *vs.* Connectionless

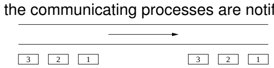
Connection-based: the processes must setup the channel before exchanging data – analogous to the telephone network;

Connectionless: the processes need not set up the channel, can exchange data immediately – analogous to mail

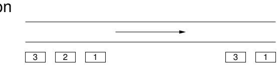
Reliability (loss of packets)

Reliable: ensure that the data sent is delivered to the respective destination

- under some assumptions;
- if not, the communicating processes are notified

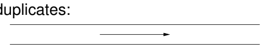


Unreliable: it is up to the communicating processes to detect the loss of messages and proceed as required by the application

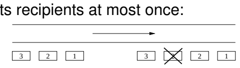


Reliability (duplication)

"Generates" duplicates: the channel may deliver duplicated messages to the destination – it is up to the recipients to detect the duplicates:

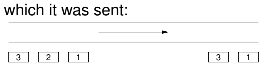


No duplicates: the channel ensures that it delivers each message to its recipients at most once:

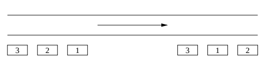


Order

Ordered: ensures that the data is delivered to its recipients in the order in which it was sent:



Unordered:



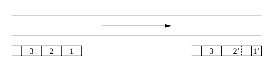
If it is important to preserve the order, it is up to the application to detect that the data is out of order and if necessary to reorder it

IMP order and reliability are orthogonal

Comunication Abstraction

Message (datagram): the channel supports the transport of messages – sequences of bits processed atomically – analogous to mail

Stream the channel does not support messages. Essentially, it works as a pipe for a sequence of bytes – analogous to Unix pipes



Other Properties of a Communication Channel

Flow control: prevents "fast" senders from overflowing with data "slow" recipients

- it does not necessarily mean that the sender has more computing power than the receiver

Number of ends of the communication channel

unicast, or point-to-point only two ends;
broadcast all nodes in the "network";
multicast subset of nodes in the network

Identification of the communicating processes

- "Name" of the process itself
- "Name" of the channel endpoint, e.g. phone number

(Process Identification on the Internet (IPv4))

Question How do identify a process on the Internet?

Answer By a pair of identifiers:

- **IP Address** :
  - Identifies the computer on the Internet;
  - It is a 32-bit, usually represented in *dotted decimal*, e.g.: `193.136.28.31`.
- **Port Number** :
  - Identifies the endpoint of a communication channel inside a computer (transport layer identifier)
  - It is a 16-bit integer (between 0 and 65535);
  - By convention, some ports are reserved for some services.

Clarification The (*IPAddress*, *PortNumber*) pair actually identifies the endpoint of the communication, the process may change

This is similar to a residence phone number: the person that answers a call may change.

Roadmap

Message-based communication

Properties of a Communication Channel

Internet Protocols
- UDP
- TCP

Multimedia Applications and End-to-end Argument

Internet Protocols

Aplication	Specific communication services
Transport	Communication between 2 (or more) processes.
Network	Communication between 2 computers not directly connected with each other.
Interface	Communication between 2 computers directly connected.

- On the Internet, the properties of the communication channel provided to an application depend on the transport protocol used (UDP or TCP):
  - The design of a distributed application depends on the properties provided by the chosen transport protocol

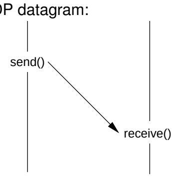
Summary of the Properties of the Internet Transport Protocols

Property	UDP	TCP
Abstraction	Message	Stream
Connection-based	N	Y
Reliability (loss & duplication)	N	Y
Order	N	Y
Flow control	N	Y
Number of recipients	1/n	1

- The abstraction provided by TCP stems from the API, or is it intrinsic to the protocol?

UDP: Observation (1/3)

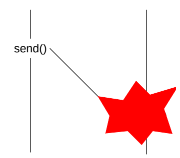
- UDP channels transport messages – UDP datagram:
  - Basically, its API supports two operations: *send()* e *receive()*;
  - Each message is transmitted by invoking `send()` once;
  - If delivered, the message will be delievered atomically, in a single invocation of `receive()`



- *Datagrams* have a maximum size of 65535 bytes:
  - Applications may have to **split** the data to send in datagrams before transmitting, and **reassemble** the data from the fragments after receiving

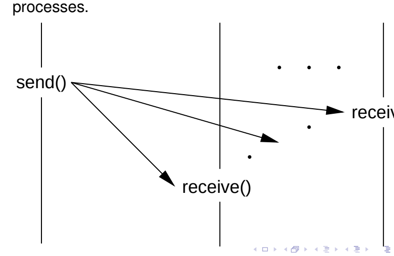
UDP: Observations(2/3)

- UDP being connectionless:
  - \+ allows a process to start transmitting data immediately;
  - – requires the specification of the the other channel endpoint on every invocation of *send()*.
- UDP provides no reliability guarantee
  - UDP datagrams may be lost or even duplicated;
  - If the application cannot tolerate the loss, or the duplication, of datagrams, it will have to detect and recover from such an event



UDP: Observations(3/3)

- UDP has no flow control
  - A receiver may be flooded with requests and run out of resources (e.g. buffers) to receive other messages.
- UDP supports *multicast*, by invoking `send()` once, it is possible to send a copy of a given message to several processes.



TCP: Observations (1/3)

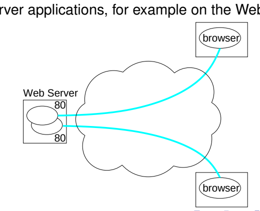
- TCP channels are **stream** channels, i.e.:
  - They are similar to Unix *pipes* on Unix, except:
    - They can be used for communication between processes in different computers;
    - They are bidirectional channels – i.e. it is possible to send data in both directions simultaneously
- Although we can also use `send()` and `receive()` to exchange data:
  - TCP does not ensure the "separation" between *bytes* sent by invoking two `send()` calls;
  - `write()` and `read()` match better TCP semantics
  - actually, the Java API uses the many "stream" classes to exchange data via TCP
- If it is essential to preserve message boundaries, it is up to the application to implement it. How?

TCP: Observations(2/3)

- TCP is connection-oriented. Communication with TCP has 3 phases:
  1. Connection set up
  2. Data exchange
  3. Connection tear down
- TCP ensures reliability (both loss and duplication of data):
  - In the event of communication problems, the connection may be aborted and the processes on its ends notified
- TCP ensures flow control
  - Prevents data loss because of insufficient resources
  - Nevertheless, TCP may be vultnerable to *denial-of-service* attacks, i.e. *SYN attacks*.

TCP: Observations(3/3)

- TCP channels have only two endpoints, supporting the communication between only two processes
- Unlike what happens with UDP, TCP channels on the same computer may have the same port number:
  - A TCP channel is identified by the pairs (IP Address, TCP Port) of its two endpoints;
  - This allows the concurrent service of several clients in client-server applications, for example on the Web:



TCP vs. UDP

Why not always use TCP?
- It provides "more" than UDP

Can you pay the cost?
- Connection must be set up before data exchange
- Recovery of a lost segment affects those that follow it
  - TCP ensures in-order delivery

Some applications cannot E.g. Internet telephony
- It is very sensitive to delays
- But can tolerate some loss

TCP provides a service that it does not need (recovery of lost data), at a cost that may be too high

Roadmap

Message-based communication

Properties of a Communication Channel

Internet Protocols
- UDP
- TCP

Multimedia Applications and End-to-end Argument

Multimedia Applications

Classes

Streaming Stored Audio and Video e.g. YouTube
- Streaming means that the contents is played before completing the reception of the entire contents

Streaming Live Audio and Video Internet radio/television
- Contents is generated as it is being sent

Real-time Interactive Audio and Video e.g. Skype
- Two-way communication

Requirements

Bandwith do not tolerate large variations

Packet delay and also its jitter (i.e. its variation) are particularly critical

Packet loss not that stringent

Internet Protocols and Multimedia Applications

- But the Internet is designed on the **best-effort** principle
  - It does not provide any guarantees, especially regarding packet-loss rate, bandwidth, packet delay and its jitter
- "Tricks people play"

  Bandwidth Use compression

  Delay and its jitter
  - For non-RT applications we can use buffering
  - For RT applications we can reduce the jitter by engineering a delay
    - And rely on the end-to-end argument

  Packet loss
  - Streaming apps can use plain TCP, as long as the buffers are large enough
  - Interactive RT apps use forward-error-correction (FEC)
    - Encoding standards (e.g. MPEG) often support FEC
- Rely on the end-to-end argument (this is explicitly referred in the article)

End-to-End Argument (around 1980)

- This is a design principle for layered systems, and states:

  If you have to implement a function **end-to-end**
  don't implement it on the lower layers
  unless there is a compelling performance enhancement

  Saltzer, Reed and Clark, "End-to-End Arguments in [...]"
- The main examples in the paper are drawn from data communication systems, but the authors also give other examples

  "For the case of the data communication system, this range includes encryption, duplicate message detection, message sequencing, guaranteed message delivery, detecting host crashes, and delivery receipts. **In a broader context, the argument seems to apply to many other functions of a computer operating system, including its file system.**"
- Why is this relevant for distributed applications?
  - Distributed applications are often layered
  - On the Internet, you have to choose between TCP and UDP
- This is a design principle, not a physics law

Dave Andersen's (?) Algorithm

Do you need everything TCP provides?
- If yes, you are done.

If not: can you pay the cost?
- If yes, you are done

If not Use UDP
- Implement what you need on top of UDP

Multicast Communication

February 16, 2021

Roadmap

Multicast Communication

- Is communication via a channel with *n* receivers
- On broadcast networks ((W)LANs), can be done efficiently by using broadcast communication provided by the MAC layer
- On point-to-point networks, can be implemented by:
  - *n* single-ended channels, if only one sender
  - $n \cdot m$ single-ended channels, if *m* senders
- But:

Multicast Communication

- Is communication via a channel with *n* receivers
- On broadcast networks ((W)LANs), can be done efficiently by using broadcast communication provided by the MAC layer
- On point-to-point networks, can be implemented by:
  - *n* single-ended channels, if only one sender
  - $n \cdot m$ single-ended channels, if *m* senders
- But:
  - The sender has to know each of the receivers
  - The sender has to send *n* separate messages
    - *n* system calls
    - Several links in the underlying communication network will be traversed by the same message
- More efficient implementations can be done at:

  Network Layer *IP Multicast*

  Application *Application Level Multicasting*
  - By means of **overlay networks**, networks built on top of the Internet

Multicast On Point-to-Point Networks

Question How can you **broadcast** efficiently on a point-to-point network?

Answer Use a spanning tree
- This is a tree that includes all nodes of the network

Question How can you **multicast** efficiently on a point-to-point network?

Answer Use a spanning tree that includes:
- The sender
- The *n* receivers
- The nodes between them (to ensure we have one tree)

IP Multicast: Lab 2

- Applications can use IP multicast via UDP
  - Specifying reliable multicast is not easy, let alone implement it
- For the sender, it is just like unicast with UDP
  - Except that it must use an IP multicast address, rather than the IP address of a host
  - The routers forward the packets along the spanning tree, so that all group members receive the datagram
- A receiver:
  1. Must **subscribe** the multicast group before receiving
     - This is needed so that it is added to the spanning tree
  2. Should **unsubscribe** the multicast group when it does not wish to receive more messages to that group
     - This allows pruning unused branches of the spanning tree

Some Alternatives

All of them use an **overlay network** whose nodes are the multicast group members, and whose edges are (virtual) links between the nodes
- Need not be a complete graph

Application-level Multicast
- Build a **spanning tree** on the overlay network
- A node that wants to send to the multicast group sends the message to the root of the tree
- The message is then multicasted, by using unicast communication along the tree branches, from the root towards the leaves

Some Alternatives

All of them use an **overlay network** whose nodes are the multicast group members, and whose edges are (virtual) links between the nodes
- Need not be a complete graph

Application-level Multicast
- Build a **spanning tree** on the overlay network
- A node that wants to send to the multicast group sends the message to the root of the tree
- The message is then multicasted, by using unicast communication along the tree branches, from the root towards the leaves

Epidemic Protocols use **limited flooding** , rather than a spanning tree, on the overlay network
- By exchanging messages with some neighbors, a message eventually spreads to all nodes in the network

Roadmap

Example: *Switch Trees* (1/4)

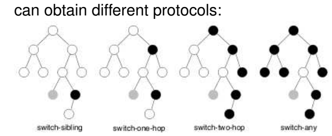
Problem The implementation of optimal algorithms such as *shortest-path tree (SPT)* or *minimum-spanning trees (MST)* is not practical
- The maximum degree of the tree nodes may easily exceed the capacity of the nodes
- The MST algorithm is complex

Idea Incrementally change the topology of the multicast tree:
- Taking into account resource constraints
- But improving some performance metric, e.g. the cost

Limitation Assumes that the multicast tree has been previously created
- For example, when a node joins the tree it becomes a child of the root
  - By performing small changes, the topology becomes more balanced

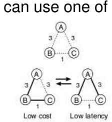
Example: *Switch Trees* (2/4)

- In principle, a node may switch its parent to any node that is not in the subtree of which it is the root. **Why?**
- By imposing restrictions on the **candidates** for new parent, we can obtain different protocols:



source: Helder and Jamin 2002

- The selection of a node among the candidates can use one of several metrics:
  - "Cost" of the tree (approx. MST)
  - Delay to the root (source) (approx. SPT)



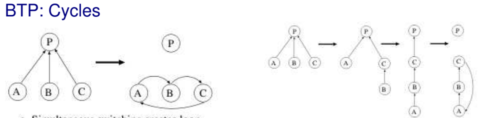
source: Helder and Jamin 2002

Example: *Switch Trees* (3/4)

Banana Tree Protocol (BTP)
- *One-hop switch* protocol
- When a node joins the multicast group, it becomes a child of the root
  - If the tree does not exist, it becomes the root
- To switch its parent, a node has to ask for permission from its new parent, which may reject the request
- If a node fails, the tree of which it is root partitions, and its children will become children of the root
  - Alternatively, to avoid overloading the root, they can become children of their grandparent, i.e. the parent of the faulty node

Example: *Switch Trees* (4/4)

BTP: Cycles



a. Simultaneous switching creates loop

b. Switching with outdated information creates loop

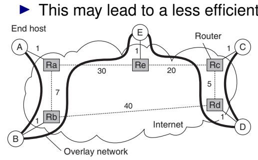
source: Helder and Jamin 2002

a Concurrent attempts by different nodes to switch their parents may lead to cycles in the "spanning tree"
- This can be avoided, if one node that is in the process to switch its parent, rejects all requests to become parent of another node
  - What's the issue with this approach?

b Outdated topological information may also lead to cycles
- May be prevented by including topological information in the authorization request
  - E.g., in the switch-one-hop algorithm, the parent of the requesting node is enough

Application-layer Multicast (ALM) Overheads

Problem Neighbors of the overlay network may be many hops away on the underlying physical network
- This may lead to a less efficient use of the network



Link stress How many times is a physical link crossed by a message on its multicast?
- For example, a message traverses *link* (Ra, Rb) twice

Link stretch Ratio of the distances between two nodes on the overlay network and on the physical network
- The cost of the path between B and C in the multicast tree is 73(pg *vs* 47 in the underlying physical network

Roadmap

Background

Objective Disseminate information by the nodes (replicas) of a distributed system

Idea Update the information by passing it to some neighbor nodes
- These will pass it on to their neighbors in a "lazy" way, i.e. not immediately.
- Eventually, all nodes with copies of that piece of information will update it.

Note The name (**epidemic**) stems from the fact that these protocols spread information/messages in a way analogous to the spread of a contagious disease

Alternatives

Anti-entropy Each node periodically chooses a random node with which it exchanges messages.

Rumor spreading A node **N** that has a "new" message passes it on to other nodes

- But if node **N** picks a node that has already received that message, it may stop disseminating that message.
- Actually, this is a variant of anti-entropy

Anti-Entropy

Idea
Periodically node P randomly chooses node Q for exchanging messages

Results from the theory of epidemic propagation

- Eventually, all nodes will receive all messages. I.e. the probability of a node missing a message tends to 0

Alternatives for Message Exchange

Push P only pushes its messages to Q

Pull P pulls in new messages from Q

Push-Pull P e Q exchange messages

- After this exchange P and Q have the same messages

Strategy Analysis

Let a **round** be the time interval required for each node to pick another node and exchange messages with it

Let p_i be the probability of a node missing a message after i rounds

A node that has not received the message after i rounds, does not receive it after $i+1$ rounds, if:

Push none of the nodes that received the message after i rounds pick it

$$p_{i+1} = p_i \cdot \left(1 - \frac{1}{N-1}\right)^{(1-p_i)N} \approx p_i e^{-1}$$

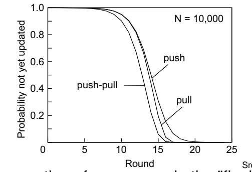
Pull it picks a node that has not received the message after i rounds

$$p_{i+1} = (p_i)^2$$

Push-Pull both

- none of the nodes that received the message after i rounds pick it, **and**
- it picks a node that has not received the message after i rounds

Strategy Comparison



Src.: van Steen & Tanenbaum

Push Propagation of a message in the "final phase" slightly slower

- As the message is disseminated, the probability of choosing a node that does **not** have the message **decreases**

Pull Propagation in the final phase slightly faster

- As the message is disseminated, the probability of choosing a node **with** the message **increases**

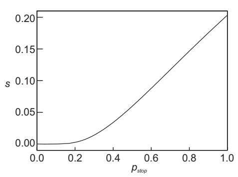
Push-Pull Combines the advantages of both

Gossiping

Idea
Variation of epidemic algorithms, in which node P looses the motivation to disseminate a message, if it tries to disseminate it to another node Q that already knows it

- Disseminates messages rather efficiently
- Does not ensure that all nodes will receive the message
  - Let p_{stop} be the probability of P stopping disseminating a message, if Q already received it
  - Then, the fraction, s, of nodes that will not receive the message is:

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



(for $p_{stop} = 0.2$, $s \approx 0.0025$) Src.: van Steen & Tanenbaum

Epidemic Algorithms: Discusssion

- Robust
  - Can easily tolerate crashes on nodes
  - Even if each node has only a partial view of the system, if this vision is continuously updated, the result is a random graph
- Highly scalable
  - Sincronization between nodes is local
- Yet, the analysis above assumes that any node can randomly select any other node
  - It would require every node to know every other node: not very scalable

RPC: Remote Procedure Call

February 24, 2021

Roadmap

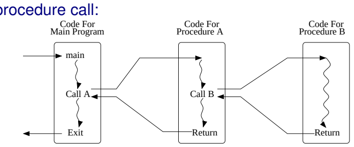
Roadmap

Remote Procedure Call (RPC)

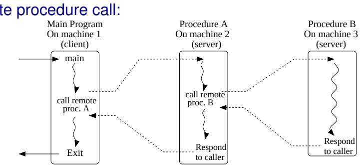
- Message-based programming with `send()/receive()` primitives is not convenient
  - depends on the communication protocol used (TCP vs. UDP)
  - requires the specification of an application protocol
  - akin to I/O
- Function/procedure call in a remote computer
  - is a familiar paradigm
  - eases transparency
  - is particularly suited for client-server applications

RPC: the Idea

Local procedure call:

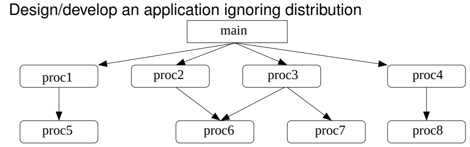


Remote procedure call:

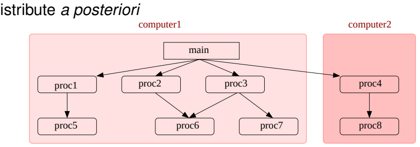


Program Development with RPCs: the Vision

- Design/develop an application ignoring distribution



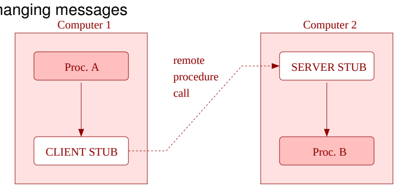
- Distribute *a posteriori*



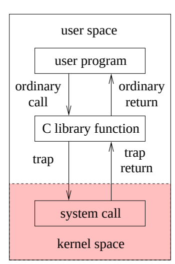
Roadmap

RPC Stub Routines

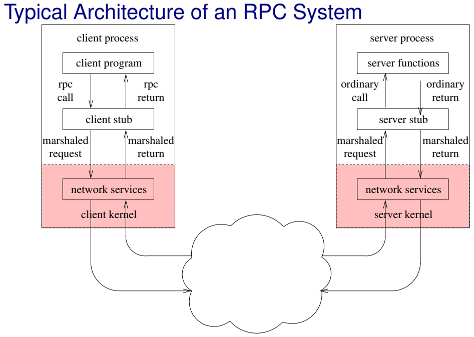
- Ensure RPC transparency
  Client invokes the **client stub** – a local function
  Remote function is invoked by the **server stub** – a local function
- The stub routines communicate with one another by exchanging messages



Well Known Trick: also Used for System Calls



Typical Architecture of an RPC System



Obs. RPC is typically implemented on top of the transport layer (TCP/IP)

Client Stub

Request

1. Assembles message: **parameter marshalling**
2. Sends message, via `write()/sendto()` to server
3. Blocks waiting for response, via `read()/recvfrom()`
   - Not in the case of **asynchronous RPC**

Response

1. Receives responses
2. Extracts the results (**unmarshalling**
3. Returns to client
   - Assuming **synchronous RPC**

Server Stub

Request

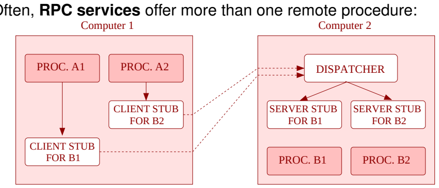
1. Receives message with request, via `read()/recvfrom()`
2. Parses message to determine arguments (**unmarshalling**
3. Calls function

Response

1. Assembles message with the return value of the function
2. Sends message, via `write()/sendto()`
3. Blocks waiting for a new request

RPC: Dispatching

- Often, **RPC services** offer more than one remote procedure:



- The identification of the procedure is performed by the **dispatcher**
  - This leads to a hierarchical name space **(service, procedure)**

Roadmap

Transparency: Platform Heterogeneity

Problems at least two:

1. Different architectures use different formats
   - 1's-complement vs. 2's complement
   - big-endian vs. little-endian
   - ASCII vs. UTF-??
2. Languages or compilers may use different representations for composite data-structures

Solution mainly two:

standardize format in the wires

+ needs only two conversions in each platform
– may not be efficient

receiver-makes-right

Transparency: Addresses as Arguments

Issue The meaning of an address (C pointer) is specific to a process

Solution Use **call-by-copy/restore** for parameter passing

+ Works in most cases
– Complex
  - The same address may be passed in different arguments
– Inefficient
  - For complex data structures, e.g. trees

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?
- Client crashes
  - Need to prevent **orphan** computations, i.e. on behalf of a dead process.

Issue A client cannot distinguish between loss of a request, loss of a response or a server crash

- The absence of a response may be caused by a slow network/server

Roadmap

RPC Semantics in the Presence of Faults (Spector82)

Question What can a client expect when there is a fault?

Answer Depends on the semantics in the presence of faults provided by the RPC system

At-least-once Client stub must keep retransmitting until it obtains a response

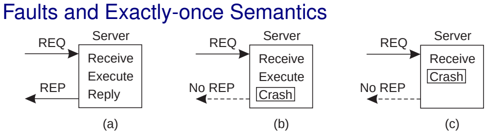
- Be careful with non-idempotent operations
- Spector allows for zero executions in case of server failure

At-most-once Not trivial if you use a non-reliable transport, e.g. UDP.

- If the RPC uses TCP, it may report an error when the TCP connection breaks

Exactly-once Not always possible to ensure this semantics, especially if there are external actions that cannot be undone

Faults and Exactly-once Semantics



Problem In the case of external actions, e.g. file printing, it is virtually impossible to ensure Exactly-once Semantics

Server policy One of two:

1. Send an `ACK` after printing
2. Send an `ACK` before printing

Client policy One of four:

1. Never resend the request
2. Always resend the request
3. Resend the request when it receives an `ACK`
4. Resend the request when it does not receive an `ACK`

Server Faults and Exactly-once Semantics

Scenario Server crashes and quickly recovers so that it is able to handle client retransmission, but **it has lost all state**

Let

`A: ACK` `P`: print `C`: crash

Fault scenarios (`ACK->P`)

1. `A->P->C`
2. `A->C(->P)`
3. `C(->A->P)`

Fault scenarios (`P->ACK`)

1. `P->A->C`
2. `P->C(->A)`
3. `C(->P->A)`

Client	Server					
	Strategy A→P			Strategy P→A		
Reissue Strategy	**APC**	**AC(P)**	**C(AP)**	**PAC**	**PC(A)**	**C(PA)**
Always	Dup	OK	OK	Dup	Dup	OK
Never	OK	Zero	Zero	OK	OK	Zero
When Ack	Dup	OK	Zero	Dup	OK	Zero
When not Ack	OK	Zero	OK	OK	Dup	OK

OK = Text printed once Dup = Text printed twice Zero = Text not printed at all

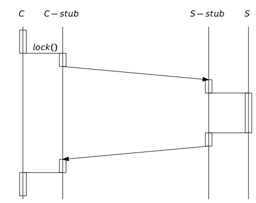
Conclusion No combined strategy works on every fault scenario

- What if server saved state on disk?

At-least-once vs. At-most-once

- Consider a locking service using two RPCs:
  `lock()`
  `unlock()`

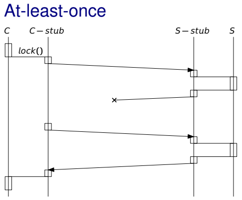
No failures and no message loss



- It does not matter the semantics supported by the RPC library

At-least-once vs. At-most-once: Lost Response

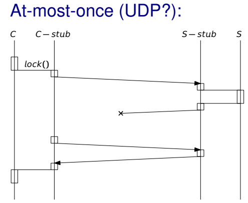
At-least-once



- Remote procedure may be invoked more than once
  - If procedure is not **idempotent**:
    - RPC must include an id as argument
    - Server must keep table with responses previously sent
  - Is `lock()` an idempotent procedure?

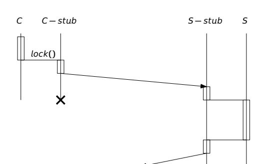
At-least-once vs. At-most-once: Lost Response

At-most-once (UDP?):



- There is no guarantee that the procedure will be executed
  - But in that case, the caller should receive an exception
- The RPC middleware ensures that the procedure is not executed more than once
  - RPC requests include an id
  - RPC system keeps table with responses
- What would be different if using TCP?

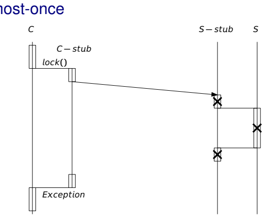
At-least-once vs. At-most-once: Client crash



- Again, the RPC semantics is irrelevant

At-least-once vs. At-most-once: Server crash

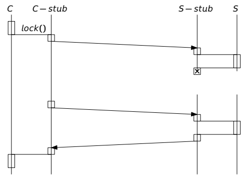
At-most-once



- Client does not know if server granted it the lock
  - Depends on when the server crashed
- Client, **not RPC**, may ask the server (or just retry)
  - Server needs to remember state across reboots
    - E.g. store locks state on disk
- Is this different from an exception upon message loss?

At-least-once vs. At-most-once: Server crash

At-least-once



- Server may run the procedure several times
  - Client stub may send several requests before giving up
- Server needs to remember previous requests across reboots (if requests are not **idempotent**). E.g.:
  - Store table request ids on disk
  - Check the request table on each request

At-least-once vs. At-most-once: Conclusions

Message loss
- At-least-once
  - Suits if requests are idempotent
- At-most-once
  - Appropriate when requests are not idempotent

Server crashes
- No clear advantage: the service itself may have to take special measures

Upon an exception can the caller tell whether the cause is message loss or server crash?

Roadmap

Further Reading

- Tanenbaum e van Steen, *Distributed Systems, 2nd Ed.*
  - Section 4.2 *Remote Procedure Call*, except subsection 4.2.4
  - Subsection 8.3.2 *RPC Semantics in the Presence of Failures*
- Birrel and Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, Pages 39-59
- A. Spector, "Performing Remote Operations Efficiently on a Local Computer Network", Communications of the ACM, Vol. 25, No. 4, April 1982, Pages 246-260
- Martin Kleppmann Part of Lecture on RPC of the Concurrent and Distributed Systems Course at the Universtity of Cambridge
- Martin Kleppmann Lecture notes of Concurrent and Distributed Systems Course at the Universtity of Cambridge
  - Section 1.3: Example Remote Procedure Cals (RPC)

Clients and Servers (Processing)

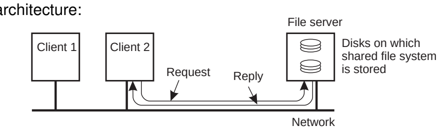
March 3, 2021

Roadmap

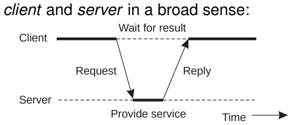
Roadmap

Clients and Servers

- Most distributed applications have a **client-server** architecture:



- We'll use *client* and *server* in a broad sense:



- A server can also play the role of client of another service.

Roadmap

Server/Object Location

Problem: how does a client find a server?

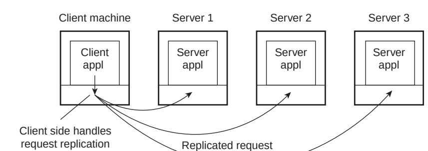
Solution: not one, but several alternatives:
- hard coded, rarely;
- program arguments: more flexible, but ...
- configuration file
- via *broadcast*/*multicast*;
- via location/naming server (later in the course)
  - local, like `portmapper` or `rmiregistry`;
  - global.

Roadmap

Distribution Transparency

Issue: Many distribution transparency facets can be achieved through client side **stubs** (also called **clerks**):

Acess e.g. via RPC;
Location e.g. via multicast;
Replication e.g. by invoking operations on several replicas:

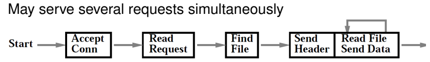


Faults e.g. by masking server and communication faults
- if possible

Roadmap

Concurrency

- There are several reasons for using concurrency:
  - Performance (+ on servers);
  - Usability (+ on clients) – still performance, really.
- The goal is to ovelap I/O with processing
- Example: Web service
  - Client-side
    - A Web page may be composed of several *objects*
    - A browser can render some objects, while it fetches others via the net.
  - Server-side
    - May serve several requests simultaneously



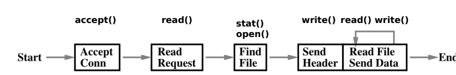
src:Pai et al. 99

How to Achieve Concurrency?

Threads
- Remember SO ...

Events
- Remember LCOM ...

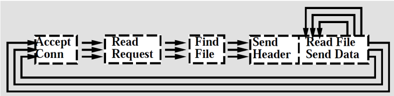
Iterative Web Server



src:Pai et al. 99

- Has only one thread
- Processes a request/connection at a time
- Each step/stage has one operation that can block
  - `stat()` is required because of the HTTP header fields `size` and `last modified`
    - But `open()` may also block
  - Server cannot process other requests while blocked
- Such a server can process only a few requests per time unit

Multi-threaded Server

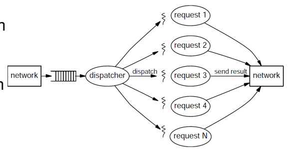


src:Pai et al. 99

- Each thread processes a request (and HTTP 1.0 connection)
- When one thread blocks on I/O
  - Another thread may be scheduled to run in its place.
- A common pattern is:

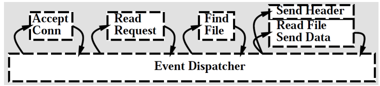
  One dispatcher thread, which accepts a connection request

  Several worker threads, each of which processes all the requests sent in the scope of a single connection



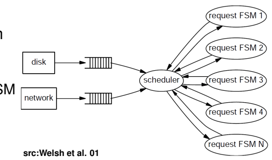
src:Welsh et al. 01

Event-driven Server



src:Pai et al. 99

- The server executes a loop, in which it:
  - waits for events (usuallly I/O events)
  - processes these events (sequentially, but may be not in order)
- Blocking is avoided by using **non-blocking** I/O operations
- Need to keep a FSM for each request
  - The loop dispatches the event to the appropriate FSM
- Known as the state machine approach



src:Welsh et al. 01

Thread vs. Event Debate

Ease of programming

Performance

Thread-based Concurrency: Ease of Programming

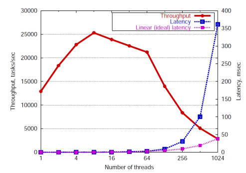
- Appears simple:
  - Structure of each thread similar to that of an iterative server
  - Need **only** to ensure **isolation** in the access to shared data structures
- Could use only monitors, e.g. synchronized methods in Java, and condition variables
  - Not so easy: there are some implications in terms of modularity (Ousterhout96)
  - Possibility of deadlocks
- Performance may suffer
  - The larger the critical sections, less concurrency
  - But the main reason for concurrency is performance

Event-based Concurrency: Ease of Programming

- Programmer needs to:
  - Break processing according to potentially blocking calls
  - Manage the state explicitly (using state machines), rather than relying on the stack
- The structure of the code is very different from that of the iterative server
- No nasty errors like race conditions, which may be elusive
- But many complain about lack of support by debugging tools
- ... and others that the it leads to poorly structured code
  - The author points out that the issue is preemption rather than multithreading
  - Actually, the problem is **lack of atomicity**
    - With multiple cores, we can have race conditions, even if there is no preemption

Thread-Based Concurrency: Performance

- Same file 8 KB reads (no disk accesss)
- No thread creation
- "4-way 500MHz Pentium III with 2 GB memory under Linux 2.2.14"

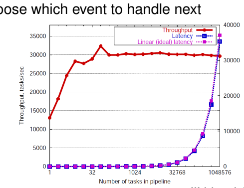


src: Welsh et al. 01

- As the number of threads increases, the system throughput increases, then levels-off and finally dives
- Clearly each thread requires some resources
- There are also issues concerning context switching
  - Actually, depends on whether user-level or kernel-level threads

Event-Based Concurrency: Performance

- Requires non-blocking (or asynchronous) I/O operations
  - Otherwise, may use multiple threads for emulation
- Allows user level scheduling
  - The dispatcher may choose which event to handle next
- Same file 8 KB reads (no disk accesss)
- Only one thread
- As the number of requests in a queue increases throughput increases until it reaches a plateau



src: Welsh et al. 01

- Needs multiple threads to achieve **parallelism** in multi core/processor platforms

TB vs EB Concurrency: Performance

- The debate was somewhat "muddled" by implementations that were less than optimal
- Actually, at the technical level this is very similar to the debate about user-level vs. kernel-level threads
- User-level threads are more efficient than kernel-level threads
  - Function calls vs. system calls
  - But performance suffers if OS does not provide non-blocking I/O
  - Worse, there are some unavoidable blocking, e.g. page faults
- Need kernel-level threads in order to take advantage of multiple processors/cores

Server Architectures

Architecture	Paral.	I/O Oper.	Progr.
Iterative	No	Blocking	easy
Multi-threaded	Yes	Blocking	races
State-machine	Yes	Non-blocking	event-driven

- To take advantage of multiple processors/cores we need to use *kernel-level threads* (or processes).
  - On state-machine designs we may use multiple threads

TB vs EB Concurrency: Conclusion

- Pure thread-based and event-based designs are the extremes in a design space
- Threads are not as heavy as processes, but they still require resources
  - You may want to bound their number
- If you want more parallelism, you need to use both:

  Threads virtually all processors now-a-days are multicore;

  Events to limit the number of threads, and therefore their overhead
- There are many frameworks supporting event-driven designs
  - Java itself offers Java NIO (non-blocking I/O)
  - Not sure about their performance
    - They are often built on top of a stack of multiple layers
  - But, often they use thread-based concurrency only by default

Thread-based Concurrency: Basic Considerations

Java
- Assume that the Java socket API is not thread-safe
  - The documentation is mute aboute this
  - Java runs on top of different OS
- You must handle concurrency explicitly

POSIX (C/C++)
- It requires many system calls, such as `accept`, `read/write`, `sendto/receivefrom`, to be **thread-safe**
  - But, data of concurrent `write`'s may be interleaved
    - I.e., `write/read` may not be **atomic** (apparently it depends on the buffer size)
- What about `send(to)/receive(from)`?
  - When used on STREAM sockets, may behave similarly to `write`
  - When used on DATAGRAM sockets, one expects POSIX-atomicity to be implied, but ...
- To be on the safe side, handle concurrency explicitly

Thread-based Concurrency: Java

Thread class/Runnable interface for creating threads
- You can use also thread pools via the interfaces `java.utils.concurrent.ExecutorService` and/or `java.utils.concurrent.ScheduledExecutorService`

Synchronized methods allow for coarse grained CC, similar to monitors

`java.utils.concurrent.locks` package for synchronization objects (locks and condition variables) to prevent race conds
- Check also the `java.utils.concurrent.Semaphore`
- Some classes of the `java.utils.concurrent` such as `ConcurrentHashMap` provide a thread-safe version of corresponding `java.utils` collection classes

Oracle's Java Tutorials' Concurrency Lesson Overview of core classes
- For a more practical oriented tutorial you can checkout `java.util.concurrent` - Java Concurrency Utilities

Event-based Concurrency with `java.nio` package

Core classes

Channels There are several subclasses

Selector For blocking waiting for more than one I/O event from a selectable channel

Buffers To read/write data from/to channels

Issue `java.nio.channels.FileChannel` is not selectable
- To avoid blockin on file I/O need to use `java.nio.channels.AsynchronousFileChannel`, which supports asynchronous I/O
  - This is more complicated than non-blocking I/O
  - There is no `java.nio.channels.AsynchronousDatagramChannel`, although one can find references to it on the Web

Getting started with new I/O (NIO) Overview of Java I/O
- Refers to non-blocking I/O as asynchronous I/O, but they are not the same
- For (an even) more practical oriented tutorial you can checkout Java NIO Tutorial

Event-driven Server Design by Doug Lea

Doug Lea's design
- This is a presentation :(
- To fill in the details check the Architecture of a Highly Scalable NIO-Based Server Blog

Roadmap

Servers and State

Problem the execution of the same task on every request may unnecessarily tax the server

Solution the server can keep some **state**, i.e. information about the status of ongoing interactions with clients;
- the size
- the processing demands

of each message are potentially smaller
- For example, in a distributed file system, the server may avoid open and close a file for each remote read/write operation
  - The server may keep a cache of open files for each client
- Depending on whether or not a server keeps state information, a server is called **stateful** or **stateless**, respectively
  - Recent cloud-related references, e.g., consider as stateful only if the state is kept in main memory

Stateless File Server

- Consider a simple file service that supports two operations:
  - read data (from file)
  - write data (to file)
- If the server is stateless it keeps no information, therefore each request must include at least:
  - operation
  - client id
  - full path name
  - file offset
  - number of bytes to transfer
  - data (only in write requests)

Upon a read request the server must:

1. Check permissions for client
2. Open the file (`open()`)
3. Set the file offset as requested (`lseek()`)
4. Read the data from the file (`read()`)
5. Close the file (`close()`)

Stateful File Server

- Server may keep information on a table about previous requests of each client (e.g.):
  - file name (or file descriptor)
  - client permissions
  - current offset
  - id of previous request
- Server may support two additional operations:
  - open file, which returns a **file handle**
  - close file
- Read/write requests need to include only:
  - operation
  - client id (possibly)
  - file handle
  - number of bytes to transfer
  - data (only in write requests)

Upon a read request the server must:

1. Look up the file handle on the table, to get the file descriptor
2. Read the data from the file (`read()`)

Stateful Servers and Failures

- Keeping state information raises some challenges:
  - of consistency;
  - of resource management;

upon failure of either clients or server

- Loss of state when a server crashes may lead to:
  - ignoring or rejecting client requests after recovery:
    - the client will have to start a new **session**
  - wrong interpretation of client requests sent before the crash:
    - TCP connection port reuse
- Keeping state (on server) when the client crashes may lead to:
  - resource depletion
    - E.g. if a client crashes before invoking `close()`
  - wrong interpretation of requests sent by other clients after the crash
    - If client id is reused (e.g. IP address and port number)

Stateful Servers and Client Crashes

Challenge resources reserved for the client may remain allocated forever

- sockets, for connection based communication
- state, in the case of stateful servers
- application specific resources

Solution **leases** (and timers):

- a server *leases a resource* to a client for only a finite time interval: upon its expiration, the resource may be taken away, unless the client **renews** the lease

Stateless Servers and Message Loss

- Stateless servers are not immune to problems arising from failures:
  - message duplication may lead to handling the same request several times
    - operations must be **idempotent**, if the transport protocol does not ensure non-duplication of packets;
    - even if the transport protocol ensures non-duplication of packets, we may still need idempotent operations
      What if the connection breaks?
- How can stateful servers handle duplicated requests?
  - Need to be careful about client identification

Stateful Servers and Client Identification

1. Use the address of the **access point**, i.e. of the channel endpoint
   - For example, the client's IP address and port
   - Issue: may not be valid for more than one transport session:
     - E.g. if a TCP connection breaks and a new one is setup in its place, the port number on the client's side may be different
2. Use a transport-layer independent **handle**. For example:
   - HTTP cookies

Servers, State and Protocols

- **Obs.-** Statelessness is a protocol issue:
  - A server can be stateless only if each protocol message has all the information for its processing independently of previous communication;
  - Likewise, a server can be stateful only if each protocol message has enough information to relate it to previous communication
- For example, Netscape had to add HTTP-header fields specifically for **cookies**.
  - HTTP is essentially stateless
    - Version 1.0 even used one TCP connection per request
  - Cookies are a device that allows a server to keep state about a client session (actually there are other types of cookies that may lead to abuse):
    - servers generate and send *cookies* to the clients
    - clients store the *cookies* received from serves
    - clients piggyback the *cookies* on HTTP requests

Roadmap

Failures

Challenges:

1. components in a distributed application may fail, while others continue operating normally
2. on the Internet it is virtually impossible to distinguish network failures from host failures or even a slow host

Solution: highly application dependent, but we'll study some general techniques

Distribution is harder than concurrency

In concurrent (local) systems the programmer needs to consider all possible execution interleavings

In distributed systems the programmer needs **also** to consider all possible failures

- Distributed systems are inherently concurrent

Roadmap

Security

Challenge: servers execute with priviledges that their clients usually do not have

Solution: servers must

authenticate clients: i.e. "ensure" that a client is who it claims to be;

control access to resources: i.e. check whether the client has the necessary permissions to execute the operation it requests.

- A related requirement is data **confidentiality**
  - need to encrypt data transmitted over the network
- Code migration (i.e. downloaded from the network) raises even more issues.

Roadmap

Communication Channel Adaptation

Order the application will have to reorder the messages (must use a sequence number), if that is important

Reliability need to use timers to recover from message loss. Have to be aware of the possibility of duplicates.

Flow control: if you want to avoid message loss because of insufficient resources

Channel abstraction: the application may have to build messages from a stream. Or, fragment messages at one end and reassemble them at the other end.

Roadmap

Further Reading

- Ch. 3 of Tanenbaum e van Steen, *Distributed Systems, 2nd Ed.*
  - Subsection 3.1.2 *Threads in Distributed Systems*, we assume the remaining material in Section 3.1 to be background knowledge (OS class)
  - Subsection 3.3.2 *Client-Side Software for Distribution Transparency*
  - Section 3.4 *Servers*
  - Section 3.2 *Virtualization*
- Arpaci-Dusseau & Arpaci-Dusseau, *Event-based Concurrency*, Ch. 33 of OSTEP book
- Pai et al., *Flash: An efficient and portable Web Server*, in 1999 Annual Usenix Technical Conference
- Welsh et al, *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*, in Symposium on Operating Systems, 2001

Introduction to Security and to Cryptography

3º MIEIC

Pedro F. Souto (`pfs@fe.up.pt`)

March 11, 2021

Roadmap

Computer Security/Cybersecurity Incidents

Ransoms

Web page defacing usually for political reasons

Online Banking Credentials Theft

Credit Card Theft and also of personal information

Denial-of-Service often tied to ransoms or political statements

Industrial/military sabotage e.g. Stuxnet

Intellectual Property Theft

Misinformation campaigns using fake news, usual for political gains, both nationally and internationally

Check
- Wikipedia's List of Security Hacking Incidents, for a list of high-profile incidents
- List of Significant Cyber Events Since 2006 by the Center for Strategic & International Studies, a USA Think Tank

Computer Security: A Definition

Security in a Computational System: *"deals with the prevention and detection of unauthorised actions by users of a computer system"*, Dieter Gollmann in Computer Security, John Wiley & Sons, 1999

- Need to specify which actions are authorized to each user (the remaining actions are unauthorized)
  - In other words, we need to specify a **security policy**, i.e. the security requirements
- Authorization requires ***authentication*** and ***access control***.
- To prevent unauthorized actions it is not always possible or may not make economic sense, in this case we will need to content ourselves with the **detection** of these actions

Security: Other definitions

- Often, security is defined in terms of ensuring:

  Confidentiality: that is, prevent unauthorized access to computer-related assets;

  Integrity: that is, prevent unauthorized modification of computer-related assets;

  Availability: that is, prevent that authorized access to computer-related assets be denied

  This is often called the **CIA triad**
- Like in Dieter Gollmann's definition, it is clear that to ensure security it is crucial to define what is authorized.

Security Process

- There are no systems 100% secure.
  - Even if this is technically possible, its economic costs may not be justifiable;
- Implementing security requires a ***risk analysis***, formal or not. This allows to identify:
  - The assets that we need to protect;
  - The threats to these assets.
- The outcome of this analysis is the specification of a security policy, i.e. of the security requirements
- To implement a security policiy, we use security mechanisms
- To verify the conformity of the implementation with the security policy, one needs to audit and to monitor system operation, usually with the help of logs.

Security Threats

Definition (ISO 27005) A potential cause of an incident, that may result in harm of systems and organization

- Internal vs. external;
- Passive vs. active;
- Or also with respect to the consequence:

  Interception e.g. snooping the communication between 2 entities;

  Interruption e.g. deny access to a Web service, via a denial of service attack

  Modification e.g. changing the contents of a message of a DB record;

  Fabrication e.g. add a *password* to an account (that should have none).
- To mitigate the consequences of a threat, so as to compy with the security policy (requirements), we need to use **security mechanisms**

Security Design

- Security cannot be implemented by adding one layer at the end of a design
  - At that time, decisions previously made may seriously restrict the options
- Some design aspects that we need to consider are:

  Layer in which layer of the computational system (e.g. network, OS, application) are security mechanisms implemented?

  Complexity vs. Simplicity shall the system have lots of functionality or is it more important to ensure high reliability?

  The unavoidable cost of reliability is simplicity.
  Anthony Hoare

  Centralization vs. Decentralization on which components does the system's security depends? I.e. what is the system's ***Trusted Computing Base (TCB)***?

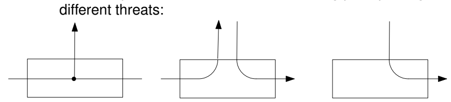
Further Reading

- Watch this great introduction to computer security of an MIT lecture by Prof. Nickolai Zeldovich
  - And, if you have time, explore the the whole course

Roadmap

Criptography

- Is one of the most used security mechanisms in distributed systems
  - Allows to protect the communication among principals against different threats:



Cryptographic Primitives

1. Encryption/Decryption algorithms
2. Cryptographic Hash Functions
3. Digital Signature Algorithms

Fundamental Principle Algorithms should be public. The security is provided by parametrizing the algorithms with ***keys***.

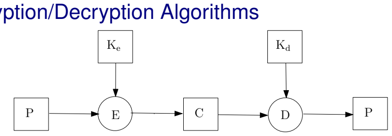
Cryptographic Systems Two:

Symmetrical (or of shared key) use a single key that is ***shared*** (K)

Asymmetrical (or of public key) use two keys one of which is ***public*** (K^+) and the other is ***private*** (K^-).

Roadmap

Encryption/Decryption Algorithms



Symmetrical, or with shared key: in this case, the keys for encrypting and decrypting are the same:

$$K_e = K_d = K$$

- The key is shared among all principals authorized to access information
- The key must be known to those principals only

Asymmetric, or with public key: in this case the encryption and the decryption keys are different:

$$K_e \neq K_d$$

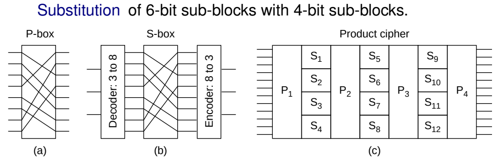
- One of these is public and the other private. Which is which?

Encryption with Shared Key: DES (1/3)

- *Data Encryption Standard (DES)* was a USA encryption standard, considered vulnerable since the mid 90's:
  - It was defeated by Moore's law, as its designers predicted
- The algorithm is relatively simple. It is based on the repeated application of 2 basic operations on bit blocks:

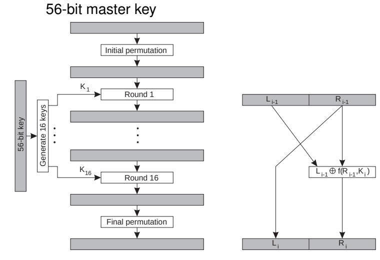
  Permutation of bits in a block;

  Substitution of 6-bit sub-blocks with 4-bit sub-blocks.



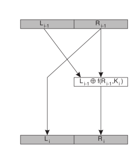
Encryption with Shared Key: DES (2/3)

- The basic algorithm operates on 64-bit blocks that are transformed in blocks with the same length;
- The encryption of a block takes 16 rounds.
  - Each round uses a different 48-bit key that is generated from the 56-bit master key



Encryption with Shared Key: DES (3/3)

- The final permutation is the reverse of the initial permutation
- The real work is performed by a *(mangler function) (f)*
  1. Expands R_{i-1} to a 48-bit block;
  2. Computes the XOR of the result with the round's key, K_i;
  3. Breaks the result in eight 6-bit sub-blocks;
  4. Each sub-block is processed by a different substitution function that converts a 6-bit block into a 4-bit block
  5. The eight 4-bit sub-blocks are combined into a single 32-bit block which is permutated



- The same algorithm is used for both encryption and decryption
- DES was replaced by AES as a US standard in 2001

Public Key Encryption: RSA (1/3)

- RSA is based on the following property of modular arithmetic:
  - Let p and q be two prime numbers;
  - Let $n = p.q$ and $z = (p-1)(q-1)$
  - Let d and e two numbers that $d.e = 1 \ mod \ z$
  - Then for any x ($0 \leq x < n$):
    $x^{d.e} = x \ mod \ n$

Public Key Encryption: RSA (2/3)

- To encrypt a message:
  1. Split it in blocks of a fixed pre-determined length, such that each block m_i, interpreted as a binary number, be less than n
  2. For each block compute:
     $c_i = m_i^e \ mod \ n$
- To decrypt an encrypted message;
  1. Split the received (encrypted) message in fixed-length blocks;
  2. Compute:
     $m_i = c_i^d \ mod \ n$
- So, to ensure confidentiality with RSA:
  - The encryption key, $K_e = (e, n)$, must be public;
  - The decryption key, $K_d = (d, n)$, must be secret;

Public Key Encryption: RSA (3/3)

- How to compute the two keys?
  1. Pick p e q, 2 very large prime numbers, e.g. $> 10^{100}$;
  2. Compute $n = pq$ e $z = (p-1)(q-1)$
  3. Select value e (surprisingly, or may be not, it can be small)
  4. Use Euclides algorithm to compute d:
     $ed = 1 \ mod \ z$
- The security of RSA relies on the dificulty of factoring a very large number (n)
  - NIST recommends 2048-bit keys
    - 3072-bit keys if security is required beyond 2030

Cipher Block Modes of Operation (1/3)

Observation The majority of the encryption algorithms encrypt fixed-size blocks (e.g. 64-bit blocks in the case of DES)
- For this reason they are known as ***block ciphers***
- ***Stream ciphers*** are another class of encryption algorithms that operate directly on sequences of bytes with an arbitrary length

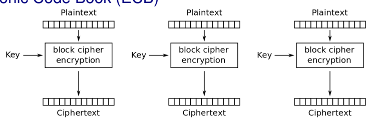
Problem How can we encrypt data/messages whose length is larger than that of a block?

Solution Just:
1. Use **padding** so that the length of the data to encrypt is a multiple of the length of a data block
2. Split the data/message in blocks and encrypt each of the blocks

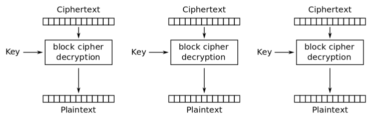
The last step can be carried out in different ways that are known as ***cipher block modes of operation***

Cipher Block Modes of Operation (2/3)

Electronic Code Book (ECB)



Electronic Codebook (ECB) mode encryption

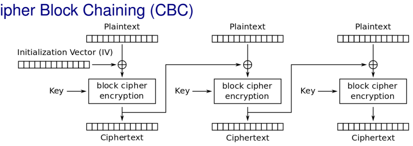


Electronic Codebook (ECB) mode decryption

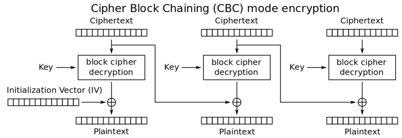
Problem Facilitates cryptanalysis
- Identical data blocks are encrypted in identical cyphered-blocks

Cipher Block Modes of Operation (3/3)

Cipher Block Chaining (CBC)



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

- The ***Initialization Vector*** is a (pseudo-)random number

Cryptographic Hash Functions

- Are used to check data integrity, among many other things
  - Someone has called them the cryptography's work horse
- Properties a cryptographic hash function, $h()$, should have:

Compression maps an input value of arbitrary length into a fixed-length hash-value;

Ease of computation

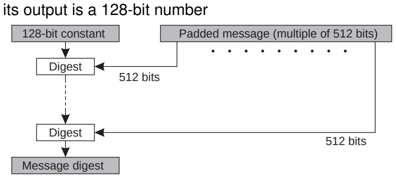
Non reversible, i.e. (*One-way*) given a hash value, y, it is computationally infeasible to find a value x such that $y = h(x)$

Weakly Collision Resistant given a value x is computationally infeasible to find a different value x' such that $h(x) = h(x')$

Strongly Collision Resistant it is computationally infeasible to find two different values x and x' such that $h(x) = h(x')$

Cryptographic Hash Functions: MD5

- The algorithm is executed in k stages, where k is the number of 512-bit blocks;
  - If necessary, the input is padded so that it length is a multiple of 512 bits.
- Each stage takes as input a 128-bit number and a 512-bit block and its output is a 128-bit number



- Each stage makes 4 passes over message block

MD5: First pass

- Each 512-bit block is split into sixteen 32-bit blocks($b_0, b_1, \ldots, b_{15}$)
- The operations performed on the first pass are:

Iterations 1-8	Iterations 9-16
$p \leftarrow (p + F(q,r,s) + b_0 + C_1) \ll 7$	$p \leftarrow (p + F(q,r,s) + b_8 + C_9) \ll 7$
$s \leftarrow (s + F(p,q,r) + b_1 + C_2) \ll 12$	$s \leftarrow (s + F(p,q,r) + b_9 + C_{10}) \ll 12$
$r \leftarrow (r + F(s,p,q) + b_2 + C_3) \ll 17$	$r \leftarrow (r + F(s,p,q) + b_{10} + C_{11}) \ll 17$
$q \leftarrow (q + F(r,s,p) + b_3 + C_3) \ll 22$	$q \leftarrow (q + F(r,s,p) + b_{11} + C_{12}) \ll 22$
$p \leftarrow (p + F(q,r,s) + b_4 + C_5) \ll 7$	$p \leftarrow (p + F(q,r,s) + b_{12} + C_{13}) \ll 7$
$s \leftarrow (s + F(p,q,r) + b_5 + C_6) \ll 12$	$s \leftarrow (s + F(p,q,r) + b_{13} + C_{14}) \ll 12$
$r \leftarrow (r + F(s,p,q) + b_6 + C_7) \ll 17$	$r \leftarrow (r + F(s,p,q) + b_{14} + C_{15}) \ll 17$
$q \leftarrow (q + F(r,s,p) + b_7 + C_8) \ll 22$	$q \leftarrow (q + F(r,s,p) + b_{15} + C_{16}) \ll 22$

  - p, q, r and s are 32-bit variables, which move from one pass to the next
    - In the first stage, p, q, r, s are initialized to pre-defined values
  - F is *F(x,y,z) = (x AND y) XOR ((NOT x) AND z)*;
    - Each of the other 3 passes uses similar functions G, H, I
  - The C_i are 32-bit constants
    - Each pass uses its own set of 16 constants, so there are 64 constants C_1 a C_{64}

Authentication with Hash Functions

- By "adding" a key to the message/data hash functions can be used to authenticate the sender and to check message integrity
  - In this case, the hash value, and also the hash function, are known as **message authentication code (MAC)**.
- In this case the hash function must satisfsy the additional property:

Computational resistance for any unknown key, k, given the values $(x, h(k,x))$ it is computationally infeasible to compute $h(k,y)$ for a different value y

Why?
- HMAC (RFC) is a MAC that ensures the same security as the hash function used:
  - MD5 is considered unsafe since 2004
- The key must be shared by all communicating ends
  - A MAC is not the same as a digital signature

Digital Signatures

- A digital signature should:
  1. Identify its author
  2. Be verifiable by others
- On a point-to-point channel, MACs allow the receiver to identify the sender, but does not allow a third party to identify the sender
  - Anyone knowing the message and the key, in principle the 2 communicating parties, nay generate an appropriate MAC

  I.e., MACs do not allow **no-repudiation**.
- Digital signature primitives are usually based on asymmetric encryption systems

Digital Signature with RSA

- Public key encryption algorithms, e.g. RSA, may be used to generate digital signatures
- In its basic form, the encrypted of a message with the senders private key can considered as a signature
  - Decryption of the encrypted message using the public key for deciphering, is the best proof that can be presented
- In practice, signing digitally comprises 2 steps:
  1. Compute the hash value of the data to sign
  2. Encrypt that hash value

  The output of the second step is a digital signature
- Not all digital signature algorithms are based in this algorithm, e.g. DSA

```
Signature sign(Message m, Key K⁻)
Boolean check(Message m, Signature s, Key K⁺)
```


Roadmap

Strength of the Cryptographic Mechanisms (1/2)

Empirically secure based on the test of time. For example DES
- There are no known vulnerabilities
- Although there is no proof of its security, it is considered secure by the cryptography community

Provably secure based on complexity theory. If its security depends on solving a problem for which no computationally feasible solution is known. For example, RSA:
- The complexity is measured in asymptotic terms: how big is sufficiently large?
- Actually, there is no proof that factoring cannot be done in polynomial time

This type of algorithm can be cracked by an attacker that has enough computing power
- It is a question of time
- ... and of keys' length

Strength of the Cryptographic Mechanisms (2/2)

Unconditionally secure based on the information theory. An algorithm is secure if an attacker cannot extract information about the plaintext by observing the ciphertext

- History shows that published cryptographic algorithms are broken mostly because of the length of the keys and not so much because of algorithm vulnerabilities
- With unpublished algorithms the history is different
  - DeCSS is may be the most recent and publicized example

The Last Word to the Experts

- *"If you think cryptography will solve your problem then you don't understand cryptography ... and you don't understand your problem."*, Bruce Schneier
- *"Cryptography is rarely ever the solution to a security problem. Cryptography is a translation mechanism, usually converting a communications security problem into a key management problem and ultimately into a computer security problem."*, Dieter Gollmann in
  Computer Security, John Wiley & Sons, 1999

Roadmap

Introduction

Criptography

Cryptographic Primitives
- Shared Key Encryption
- Public Key Encryption
- Cryptographic Hash Functions
- Digital Signatures

Conclusion

Further Reading

Further Reading

- Chapter 9, Tanenbaum e van Steen, *Distributed Systems, 3rd Ed.*
  - Section 9.1: *Introduction to Security*

Secure Communication Channels

March 21, 2021

Roadmap

Introduction

Shared Key Authentication Protocols

Public-key Authentication Protocols

Session Keys

Secure Channel Implementation Layer

Key Management

Further Reading

Roadmap

Introduction

Shared Key Authentication Protocols

Public-key Authentication Protocols

Session Keys

Secure Channel Implementation Layer

Key Management

Further Reading

Secure Communication Channels

- Many network security problems can be mitigated with the help of **secure channels**, which can guarantee:
  - Authentication of the communicating parties, i.e. that the entities at the ends of the channel are who they claim to be
  - Integrity/Authenticity i.e. that the messages were not modified in transit
  - Confidenciality i.e. that an attacker cannot observe the contents of a message
- Usually, integrity or confidentiality do not make sense without authentication
  - And authentication does not make much sense without integrity

Authentication

- Usually, during the setting up of a secure (communication) channel the two entities authenticate each other
  - Actually, on the Web, most often only one of the entities is authenticated
- Often, the channel set up phase includes also the establishment of a **session key** that is used to ensure integrity or confidentiality
- Passwords are not appropriate for authentication while setting up a secure channel
  - Instead, one often uses *challenge/response* protocols

Roadmap

Introduction

Shared Key Authentication Protocols

Public-key Authentication Protocols

Session Keys

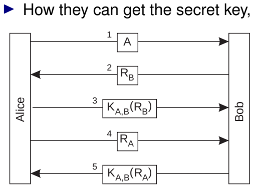
Secure Channel Implementation Layer

Key Management

Further Reading

Shared Key Authentication Protocols

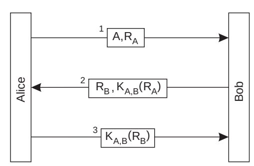
Assumptions The parties (Alice/A e Bob/B) at the two ends share a secret key ($K_{A,B}$)
- How they can get the secret key, will be discussed shortly



- Messages 2 and 3 allow B to authenticate A;
- Messages 4 and 5 allow A to authenticate B;

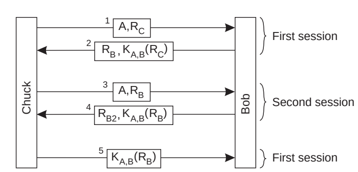
"Optimized" Authentication Protocol with Shared Key

- We could ***optimize*** this protocol as follows:



- But this is vulnerable to a **reflection attack**.

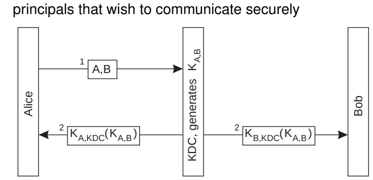
Reflection Attack



- The problem is that the 2 parties use the same challenge in two different executions
- A principle to avoid reflection attacks is to make the protocol asymmetrical:
  - E.g. require one party to use an odd challenge and the other one an even challenge

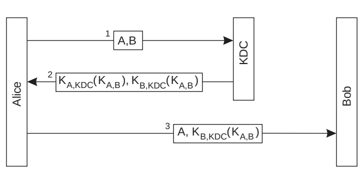
Mediated Athentication (with KDC) (1/2)

- Shared key authentication (without a KDC) is not scalable:
  - Each pair of principals must share a secret key.
- A solution is to use an mediator, the *Key Distribution Center (KDC)*, in which all principal must **trust**.
  - The KDC shares a secret key with each principal
  - Upon request, the KDC generates secret keys for sharing between principals that wish to communicate securely



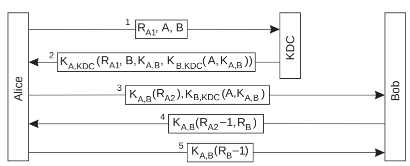
Mediated Athentication (with KDC) (2/2)

- What if B receives A's first message to B, before it receives the key from the KDC?



- This protocol is not complete:
  - A e B must authenticate mutually
    - I.e. prove that they know $K_{A,B}$

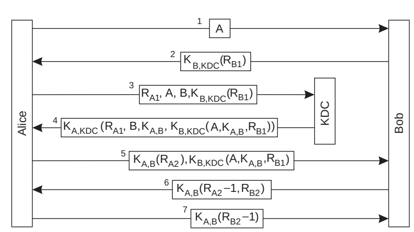
Needham-Schroeder's Protocol (1/2)



- The *nonce* (R_{A1}) is used to ensure that A comunicates with the KDC (it prevents *replay attacks*).
- The KDC includes B's identity in the response, to prevent C from impersonating B, by replacing B with C, in message 1.
- Messages 3 and 4 allow A to authenticat B
- Messages 4 and 5, allow B to authenticate A
- $K_{B,KDC}(A, K_{A,B})$ in message 2 is known a the *ticket to Bob*

Needham-Schroeder's Protocol (2/2)

- In 1981, Denning and Sacco found a vulnerability, if C learns A's key ($K_{A,KDC}$), even if it has been replaced:
  - In this case, C may impersonate A, in its communication with B
- In 1987, Needham and Schroeder published a new version of the protocol that fixed that vulnerability



Roadmap

Introduction

Shared Key Authentication Protocols

Public-key Authentication Protocols

Session Keys

Secure Channel Implementation Layer

Key Management

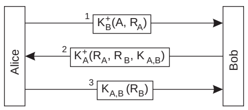
Further Reading

Public-key Authentication

Assumptions

1. The principals (Alice/A and Bob/B) know the public keys of one another
   - Below, we discuss how one can learn the public key of a principal
2. Private keys are *known exclusively* by the respective principal

Basic Protocol

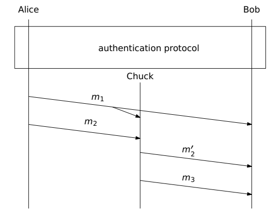


- In addition to authenticate both principals, this protocol also negotiates a **session key**
  - The share key can be used to ensure confidenciality and integrity

Roadmap

Data Integrity/Confidentiality (1/2)

- Authentication upon setting up a secure channel is not enough
  - If after authentication no measures are taken, i.e. the messages are exchanged in the clear, an attacker can not only intercept these messages but also modify them or even fabricate them



- To ensure authenticity/integrity and confidentiality of the messages exchanged after authentication, A and B can use cryptography

Data Integrity/Confidentiality (2/2)

- Although encrypting a message can ensure confidentiality, it may not be sufficient to ensure integrity
  - In secure communication, by **integrity** we mean that the channel should be able to detect modification of the messages
  - But the secure channel may not have enough context information to determine whether or not the message has been modified

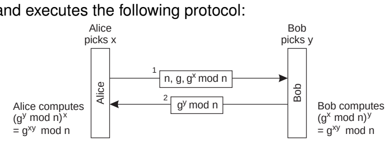
  One needs to use **authenticated encryption**
  - An obvious approach is **Encrypt-then-MAC (EtM)**, i.e. first encrypt the message then compute a MAC of the ciphtertext (note that different keys should be used)
  - But there are approaches that use a single key

Session Key

- To ensure confidentiality, it is important to use a **session key** that is different from the key used for principal authentication:
  - Public key encryption is less efficient the shared key encryption
  - Keys *wear out* with use: the more ciphertext an attacker has, more likely it is she will succeed finding the key
  - The use of the same key over multiple sessions, makes *replay attacks* more likely to succeed
  - If a session key is compromised, the attacker will not be able to decrypt messages exchanged in other sessions
- Actually, most secure channels have provisions to change keys in the middle of a session
  - This prevents compromising the key or replay attacks in long running sessions

Diffie-Hellman Key-Agreement Protocol (1/2)

- Let *n* be a large prime and *g* a number less than *n* with some properties (for additional security)
  - These number must be known *a priori*, and may be public
- Each principal chooses a private and secret large number, *x* and *y* and executes the following protocol:



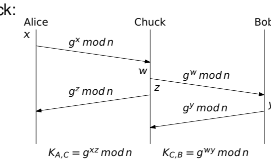
- The session key can be computed as $g^{xy} \bmod n$

  Question Why can't an attacker overhearing the communication do the same?

  Answer It would have to compute the discrete logarithm, which is considered computationally tractable
  - However, there are known efficient algorithms for special cases

Diffie-Hellman Key-Agreement Protocol (2/2)

- As described, the DH protocol is vulnerable to a man-in-the-middle attack:



- To defend it against such an attack, we can use:

  Published DH numbers e.g. *A* would publish $g^x \bmod n$ somewhere, and always reuse *x* for computing the session key (same for *B*)

  Authenticated DH i.e. messages of the DH-protocol are authenticated to prevent tampering. This requires:
  - Either a secret key shared among *A* and *B*
  - Or a public-keys (and private) for *A* and *B*

Perfect Forward Secrecy

Question Why do we need yet another protocol (DH)?
- It is possible to generate a secure session key from the nonces, i.e. random numbers, that are usually exchanged by the authentication protocols
  - If we follow certain rules

Answer **Perfect forward secrecy**, i.e. an attacker will not be able to decrypt a recorded session even if (s)he later
- Breaks into both *A* and *B*
- Steals their long-term secrets

as long as *A* and *B* delete their secret numbers (*x* and *y*, respectively)

Watch out Schor has invented polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer
- DH and RSA will become insecure when quantum computing becomes practical
  - According to the Wikipedia, the largest integer factored using a quantum computer was 291 311, a 19-bit value, in 2017

Session Key for Unidirectional Authentication

- The Web (SSL/TLS) uses mostly public key cryptography **unidirectional** authentication:
  - Clients authenticate Web servers using public key cryptography
  - Servers do not authenticate clients
    - Public key management at Internet scale is nof easy
- In this case, the session key can be computed as follows:
  1. The client can generate (randomly) the session key and send it to the server encrypted with the latter's public key
  2. Client and server can execute Diffie-Hellman, but only the server's messages are authenticated
- In any case:
  - The client is guaranteed (under some assumptions) that it has set up a secure channel with the server
  - The server
    - has **no** idea who is on the other end of the channel
    - is guaranteed that on the other end of the channel is always the same client

Roadmap

Secure Channel Implementation Layer

- In principle, it is possible to implement a secure channel at any layer of the communication stack:

  Data Link e.g. Wi-Fi Protected Access (WPA)
  - Protects only the communication in one "segment".

  Network - e.g. IPSec
  - Usually this is implemented by the OS
  - Uses IP addresses for identification – and authentication

  Transport - e.g. SSL/TLS
  - Requires applications to be modified – (*sockets*);

  Application - e.g. `ssh`, SMIME, PGP
  - Protects application-specific *objects*, e.g. email messages stored on servers

Roadmap

Key Distribution

Problem How to get the keys required by the authentication protocol?

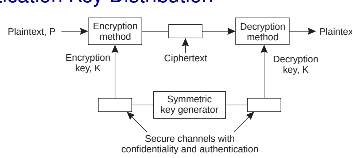
- All protocols assume that a principal knows a key bound to the other principal

  Shared key in the case of symmetrical cryptographic systems;

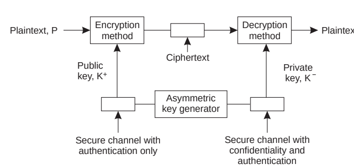
  Public key in the case of asymmetrical cryptographic systems

Solution Depends on the type of cryptographic system

Authentication Key Distribution



(a)



(b)

Public Key Certificates (1/3)

- The challenge with shared keys is to ensure that they are kept secret
- The challenge with public keys is to ensure the binding between a public key and a principal
  - Cryptographic protocols can only check whether a public key matches a private key
  - If *C* convinces *A* that *B*'s public key is a key matches a private key *C* knows, then *C* can impersonate *B*
- The solution to address this challenge is based on **public-key certificates/digital certificate**, which contain:
  1. The subject's (principal's) name
  2. A public key that matche's the subject's private key
  3. A signature of the remaining information by a **Certification Authority (CA)**
  4. The name of the CA

Public Key Certificates (2/3)

- The assumption is that the CA's public keys are well known
  - Web browsers are shipped with the public keys of (too) many CAs
- When a browser validates/accepts a certificate, the user **trusts** that the principal's it wants to communicate with has the public key in the certificate
  - But CAs can be tricked into issuing certificates for someone else to attackers (e.g. Verisign issued a certificate for Microsoft)
- On the Web, the trust model used by digital certificates is not only oligarchic but also hierarchical
  - A CA can delegate on other CAs, i.e. issue a certificate vouching for their trustworthiness
- PGP uses the "anarchy model"
  - Each user can choose which public-keys it trusts (**trust-anchors**)
  - A user can sign certificates for anyone else
  - To get a key's certificate, one needs to find a path starting on some trust anchor
- A key issue in this scheme is naming: is the John Smith whose key I need the John Smith in some certificate?

Public Key Certificates (3/3)

- Digital certificates have expiration dates, often of several months or even years:
  - Browsers typically warn the user when a server presents an expired certificate

Problem What if the private key is compromised before the certificate expires?

Solution Revoke the certificate
- CAs periodically publish **Certificate Revocation List (CRL)** with certificates that have been revoked.
  - What should the period be?
  - To reduce the amount of data transferred, CAs can publish a full CRL with a larger period and delta CRLs with a shorter period
- The Online Certificate Status Protocol (OCSL) (RFC 2560) allows browsers to verify the validity of a certificate in real-time
  - If the CA runs an on-line revocation server

Roadmap

Further Reading

- Chapter 9, Tanenbaum e van Steen, *Distributed Systems, 3rd Ed.*
  - Section 9.2: *Secure Channels*
  - Subsection 9.5[.1]: *Key Management*

Fault Tolerance

Introduction

Pedro F. Souto (pfs@fe.up.pt)

March 23, 2021

Fault Tolerance

Definition A system/component **fails** when it does not behave according to its specification.

Definition A system is **fault-tolerant** if it behaves correctly despite the failure of some of its components
- Obviously, no system tolerates the failure of **all** its components
- Usually, a system tolerates only some kinds of failures, as long as they do not occur too frequently or they only occur on some of its components

Observation Fault tolerance is achieved by design. We need to include some redundancy in the system:

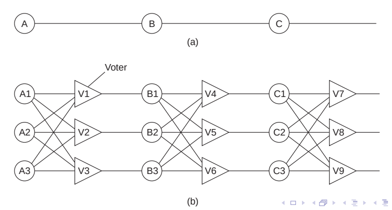
HW Processors, memory, I/O devices, communication links, ...

Time For executing additional tasks, e.g. retransmission of a packet

SW To manage the redundant HW, or the repetition of task or even n-version programming

Triple Modular Redundancy (TMR)

- Well-known HW-based FT-technique, proposed by von Neumman
- Each node is triplicated and works in parallel
- The output of each module is connected to a voting element, also triplicated, whose output is the majority of its inputs
- The configuration can be applied to each stage of a chain
  - It masks the occurence of one failure in each stage
  - What if a voter fails?



FT and Distributed Systems

Obs. Unless a distributed system is fault tolerant it will be less **reliable** than a non-distributed system
- A distributed system comprises more components than a non-distributed system
- In the 1980's, Lamport famously wrote in an email message: *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*

Obs. The inherent HW redundancy in a distributed system makes it particularly suitable for making it fault tolerant
- But, fault-tolerance does not emerge directly from distribution, it must be engineered

Reliability and Availability

Reliability ($R(t)$) the probability that a system **has not failed until** time t
- Particularly important for **mission**-oriented systems, such as spacecrafts, aircrafts or cars
- It is often characterized by the **mean time to failure (MTTF)**

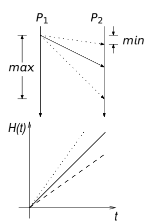
Availability Assumes that a system may be repaired after failing.

Limiting the probability that a system is working correctly: $\alpha = \frac{MTTF}{MTTF+MTTR}$, where *MTTR* is the mean time to repair
- Particularly important for systems like utilities, services on the web, that tolerate the occurrence of failures

Obs. Reliability and availability are somewhat orthogonal:
- A system A may be more reliable than system B and still be less available than system B
- A system A may be more available than system B and still be less reliable than system B

Distributed System Model

- A set of sequential processes that execute the **steps of a distributed algorithm**
  - DS are inherently concurrent, with real parallelism
- Processes communicate and synchronize by exchanging messages
  - The communication is not instantaneous, but suffers delays
- Processes may have access to a local clock
  - But local clocks may drift wrt real time



- DS may have partial failure modes
  - Some components may fail while others may continue to operate correctly

Fundamental Models

Synchronism characterizes the system according to the temporal behavior of its components:
- processes
- local clocks
- communication channels

Failure characterizes the system according to the types of failures its components may exhibit

Models of Synchronism

Synchronous iff:
1. there are known bounds on the time a process takes to execute a step
2. there are known bounds on the time drift of the local clocks
3. there are known bounds on message delays

Asynchronous No assumptions are made regarding the temporal behavior of a distributed system
- These 2 models are the extremes of a range of models of synchronism

Dilemma
- It is relatively simple to solve problems in the synchronous model, but these systems are very hard to build

Failure Models (1/2)

Characterize a system in terms of the failures of its components, i.e. the deviations from their specified behavior

Crash a component behaves correctly until some time instant, after which it does not respond to any input

Omission a component does not respond to some of its inputs
- Loss of a message can be seen as an omission failure of the communication channel or of either processes at the channel ends

Timing/Performance a component does not respond on time, e.g. it may respond too early or too late
- Makes sense only on synchronous systems. Why?

Byzantine/arbitrary a component behaves in a totally arbitrary way
- For example, a process may send a message as if it were another process

Failure Models: Taxonomy (2/2)

Crash-Recovery In this model, we assume that a faulty process may crash and recover a **finite** number of times
- The practice is that if nothing is stated, then **they do not**

Failure and Synchrony Models
- The byzantine model is similar to the asynchronous model in that:
  - Neither model makes any assumption wrt the aspect of behavior it is supposed to describe
- In the absence of faults, the synchronous and the asynchronous models are **equivalent**
  - They can solve the same set of problems

Further Reading

- Section 8.1: Introduction to Fault Tolerance, Tanenbaum and van Steen, *Distributed Systems*, 2nd Ed.

Elections

Pedro F. Souto (pfs@fe.up.pt)

April 17, 2021

Leader Election

Why Many distributed algorithms rely on a process that plays a special role – **coordinator/leader**. Such algorithms usually are:
- Simpler
- More efficient

What Upon completion of the algorithm all non-faulty nodes agree on who the coordinator is.
- Only one node is elected the coordinator
- All nodes know the identity of the coordinator

Garcia-Molina's Algorithms: Introduction

- The algorithms were proposed in the scope of **system reorganization** upon failure/recovery of system components. But, elections are also useful:
  - At initialization;
  - To add/remove nodes (to a less extent).
- GM observes that we can ensure fault-tolerance by means of two approaches:
  - By masking failures i.e. by using algorithms that continue to work correctly, even if some system components fail:
    - This is the only approach if we need continuous operation
    - Also likely to be the more appropriate, if failures are common
  - By reorganizing the system i.e. by halting normal operation and take some time to reorganize the system
    - Likely to be allow simpler algorithms
- We abstract the leader election problem from this context
  - This leads to simpler versions of GM's algorithms

Some notes on the paper

This paper is really worth the reading

- It is very well written
- It is an early paper on distributed algorithms and GM explains the issues at length
- It touches on several recurrent issues in distributed systems/algorithms:
  - Fault-tolerance
  - Synchronous vs asynchronous systems
  - Failure detection (and its impossibility in asynchronous systems)
  - Groups of processes
  - RPCs
- GM is very careful/rigorous:
  - Assumptions
  - Specifications
  - Algorithms
- And, in spite of all that, the specification for asynchronous systems is buggy

System Model/Assumptions

1 All nodes cooperate and use the same algorithm

4 All nodes have some **stable(/safe)** storage

5 When a node fails, it immediately halts all processing.
- Crashed nodes may recover
- Data on stable storage is not lost, i.e. is as before the crash

3 The communication subsystem does not spontaneously generate messages

6 There are no transmission errors (but messages may be lost)

7 Messages are delivered in the order in which they are sent

8 The communication system does not fail and has an upper bound on the time to deliver a message, T

9 A node always responds to incoming messages with no delay

Observation Assumptions 8 and 9 mean the system is synchronous.
- The author claims that they are reasonable both for a LAN or a high-connectivity network
- They will be dropped below

Specification: State

- Virtually all distributed algorithms may be described by state machines:
  - Describing the operation of each node (process)
  - Changing their state in response to reception of messages or to the passage of time

$S(i).s$ state of the node i: one of DOWN, ELECTION and NORMAL [a]
- When a node crashes its state changes automatically to DOWN

$S(i).c$ the coordinator according to node i

[a] G-M considers an additional state, but here we are presenting election algorithms independently of their application

Specification of Leader Election

Assertion 1 At any time instant, for any two nodes, if they are both in NORMAL state, then they agree on the coordinator:

$$\forall_{i,j} : S(i).s = S(j).s = \text{NORMAL} \Rightarrow S(i).c == S(j).c$$

Assertion 2 If no failures occur during the election, the protocol will eventually transform a system in any state to a state where:

a) there is a node i such that $S(i).s = \text{NORMAL}$ and $S(i).c = i$

b) all other non-faulty nodes $j \neq i$ have $S(j).s = \text{NORMAL}$ and $S(j).c = i$

Safety vs. Liveness Properties (Parenthesis)

- Any specification can be expressed in terms of **safety** and **liveness** properties (Lamport 77):
  - Safety property states that something (bad) will not happen
    - Proving such a property involves proving an invariant
    - Once an execution violates a safety property, there is nothing that can be done to fix that
  - Liveness property states that something (good) must happen
    - Proving such a property involves a different technique
    - Any "partial" execution can always be extended so that eventually something good happen
      - If that is not possible, then something bad must have happened, i.e. some safety property must have been violated
- What about the properties of the leader election specification?

Leader Election vs. Mutual Exclusion

1. In an election fairness is not important
   - All we need is that one node becomes the leader
2. An election protocol must deal properly with the failure of the leader
   - Usually, mutual exclusion protocols assume that a process in a critical section does not fail
3. All nodes need to learn who the coordinator is

The Bully Election Algorithm (1/3)

Idea A node wishing to become a leader:
- Looks around to ensure stronger nodes are not up (**phase 1**)
- If does not see any, it
  1. imposes itself as leader; (**phase 2**)
  2. brags about it.

Convention The smaller a node's identifier the **stronger** it is [a]

[a] G-M uses the inverse order

The Bully Election Algorithm (2/3)

Phase 1 A node wishing to become leader checks if stronger nodes are around sending them an ARE-U-THERE message
- If present, a stronger node responds with a YES message and initiates a new election itself
  - By checking if stronger nodes are around
  - What if the challenged node is already in the middle of an election?
- A candidate whose challenge is answered, backs off
  - But should start a timeout to detect a possible failure of the challenger

The Bully Election Algorithm (3/3)

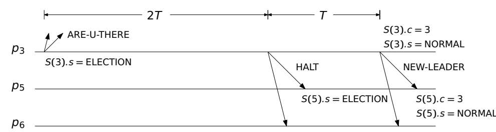
Phase 2 Node i begins phase 2, if it does not receive any response to its probe within $2T$. It comprises two steps:
1. Sends a HALT message to weaker nodes
   - Upon receiving HALT, a node
     - 1.1 Sets its state to ELECTION
     - 1.2 Cancels any election it has already started
2. T time units later, node i sends a NEW-LEADER message to weaker nodes, and sets $S(i).c$ to i and $S(i).s$ to NORMAL
   - Upon receiving that message, node k sets $S(k).c$ to i and $S(k).s$ to NORMAL

Comment The HALT message (1st step) is required to ensure that **Assertion 1** is **not** violated

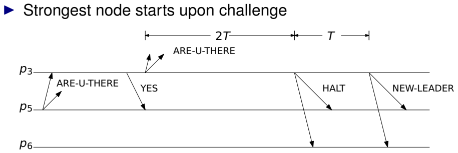
Failure of the candidate during the second phase will trigger a new election

Bully Algorithm: Example Execution

- Strongest node starts upon failure detection



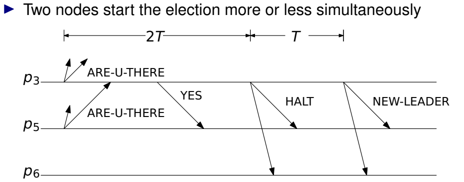
- Strongest node starts upon challenge



Food for thought Is it possible to remove the HALT message?

Bully Algorithm: Concurrent Executions (1/3)

- Two nodes start the election more or less simultaneously



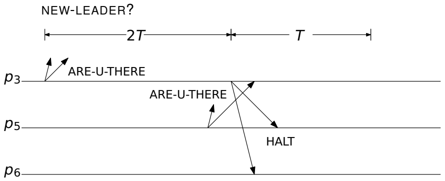
Stronger nodes will reply to weaker ones

Strongest node will finish its election
- unless it fails

Weaker nodes will back off

Bully Algorithm: Concurrent Executions (2/3)

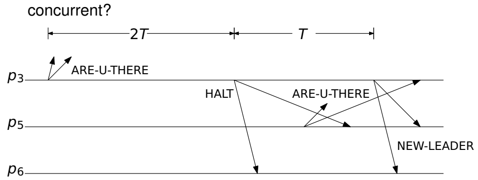
- What if the HALT and ARE-U-THERE messages are concurrent? I.e.
  - What if ARE-U-THERE is received between HALT and NEW-LEADER?



  - Should p_3 send YES?
  - Should p_3 start another election?
  - Imagine a run where p_5 has just recovered, and did not receive HALT

Bully Algorithm: Concurrent Executions (3/3)

- Is it possible that the ARE-U-THERE and NEW-LEADER be concurrent?



The Bully Election Algorithm and Failures

What about node failures? Depends on the node:
- Leader Upon detection of failure of the current leader
  - A process initiates an election
- Candidate Upon detection of failure of the candidate
  - A process initiates an election
- Other processes it does not matter
  - GM's algorithm starts a new election on such an event, because its focus is on reorganization

What about recovery of a node (after a failure)?
- The node initiates an election

What if a another node has already initiated an election?
- This is just another case of a concurrent execution – see above
  - Execution depends on which node is stronger

The Actual Bully Algorithm (1/3)

RPC rather than messages, with some interesting features
- Syncronous caller is suspended until it receives the reply from remote process or there is a timeout
- ONTIMEOUT clause (argument is time bound to receive reply, T)
  - ARE-U-THERE, HALT just sends the next RPC (synchronous)
  - NEW-LEADER restart election (reorganization)
- Immediate assume special implementation (without context switch nor RPCs on remote side) for low latency
  - Used for RPCs that use timeouts (all of them)

Failure Suspector each node has a failure suspector that triggers a TIMEOUT event, if the node does not receive a message from the **last known leader** for some time
- Upon a TIMEOUT event
  - If s == *NORMAL* the node calls ARE-U-THERE
    - And starts a new election if ARE-U-THERE times out
  - Otherwise starts a new election immediately

The Actual Bully Algorithm (2/3)

ELECTION is a procedure that executes the entire protocol by calling appropriate RPCs (?sequentially?). E.g.:

ARE-U-THERE is a null RPC, i.e. it has no statements
- Receiver does not initiate election if stronger.
- Candidate gives up, if ARE-U-THERE returns

HALT nodes that respond are added to the candidate's Up set, which belongs to the process state

NEW-LEADER Is sent only to nodes that are in Up
- If some node does not reply, candidate starts new election

Recovery is a procedure that is executed automatically upon recovery of a node, and starts a new election (call ARE-U-THERE RPC ...)
- If it is stronger than the current leader, the latter will not receive ARE-U-THERE, but it will move to the ELECTION state upon reception a HALT RPC and

Question How is a weaker recovering node integrated in the system?
- See the CHECK procedure below

The Actual Bully Algorithm (3/3)

Reorganization (not covered earlier)

Check is a procedure executed periodically by the leader
- It scans every other node (using the ARE-U-NORMAL RPC) to find out if there are new processes
- It has a dual purpose:
  1. Find new nodes
  2. Allows other nodes to detect failure of the leader

ARE-U-NORMAL RPC checks the state of a remote node
- The response is: $s ==$ NORMAL ? YES : NO
- Upon reception of a NO, the caller starts a new election

Leader Election without Assumptions 8 and 9

If we drop assumptions:

8 The communication system does not fail and has an upper bound on the time to deliver a message, T

9 A node always responds to incoming messages with no delay

Then assertions:

Assertion 1 At any time instant, for any two nodes, if they are both in NORMAL state, then they agree on the coordinator

Assertion 2 If no failures occur during the election, the protocol will eventually transform a system in any state to a state where there is a coordinator

cannot be satisfied always:
1. Assume node i is the coordinator, has not crashed but it does not respond to other nodes because it is too slow
2. From the point of view of other nodes, it has crashed, so to satisfy Assertion 2, they must elect a new coordinator
3. But, if they elect a new coordinator, Assertion 1 will be violated

Specification without Assumptions 8 and 9 (1/2)

Groups

Definition Is a set of nodes with a **group id**, i.e. an identifier
- All messages are tagged with the group id
- Not all messages with foreign group ids can be ignored

Node state Includes also the following pieces of information:

$S(i).g$ the current group id;

Specification without Assumptions 8 and 9 (2/2)

Assertion 3 At any time instant, for any two nodes i and j in NORMAL state and in the same group, then they must agree on the coordinator: $S(i).s =$ NORMAL $\wedge S(j).s =$ NORMAL $\wedge S(i).g = S(j).g \Rightarrow S(i).c = S(j).c$
- This alone is weak, as it can be satisfied by a singleton group

Assertion 4 Suppose that:
1. there is a set of operating nodes R which all have two way communication with all other nodes in R. That is Assumptions 8 and 9 hold for nodes in R
2. there is no superset of R satisfying the previous property
3. no node failures occur during the election

then the election algorithm will eventually transform the nodes in set R from any state to a state where there is i in R such that for every node j in R:

$$S(j).s = \text{NORMAL} \wedge S(j).c = i \wedge S(j).g = S(i).g$$

Note Assertions 1 and 2 are special cases of Assertions 3 and 4

The Invitation Algorithm (1/2)

Idea Rather than imposing itself as a leader, a node wishing to become a coordinator invites others to join in a group where it is the coordinator
- Initially, each node creates a singleton group, of which it is the coordinator
- Periodically, coordinators try to merge their group with other groups in order to form larger groups

Description

Failure detection a node that is not a leader periodically checks if its leader is still alive (using ARE-U-THERE messages)
- If not, it creates a singleton group of which it is the leader

Group merging a node that is a leader periodically probes all other nodes for leadership (using ARE-U-LEADER messages)
- If one or more nodes reply, node i initiates the merging protocol after a delay inversely proportional to its priority
  - The variable delay helps preventing different nodes to initiate the merging concurrently

The Invitation Algorithm (2/2)

1. Node i, leader and candidate, sends an INVITATION message:
   - to all leaders that have responded
   - to the members of its current group
2. When a leader j receives an INVITATION, it forwards it to the other group members
3. All nodes that receive an INVITATION, directly or indirectly, respond with an ACCEPT message to the candidate (to leader)
4. The candidate adds the sender of each ACCEPT message as group member
5. After enough(?) time to receive ACCEPT messages from all group members, the new leader sends a READY message to all of them
   - Note that assumptions 8 and 9 hold for R
6. Upon receiving the READY message to a previously sent ACCEPT, node k joins the new group and sends an ACK message
   - If a node does not receive a READY message after a timeout, it initiates a new election

The Invitation Algorithm: Concurrency

- A process moves to the ELECTION state, before:
  - Sending the INVITATION message, if candidate
  - Sending the ACCEPT message, otherwise
- A process moves to the NORMAL state, after:
  - Sending the READY message, if candidate
  - Receiving the READY message, otherwise
- A process responds to:
  - ARE-U-LEADER
  - INVITATION

  messages only if its state is NORMAL, i.e. it is not participating in an election:
  - A process does not participate in more than one election at a time

The Invitation Algorithm: Liveness

- Process failure is suspected using timeouts while waiting for the reception of some messages in response to:
  - ARE-U-THERE, sent to the coordinator
  - INVITATION, sent to other leaders (or by the leaders to their group members)
  - ACCEPT, sent to candidate
  - READY, sent to the new group members
- Upon timeout, there are several possible actions:

  Advancing to the next phase of the protocol
  - E.g. when waiting for responses to INVITATION

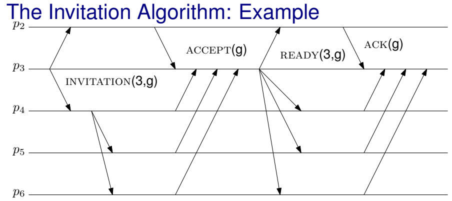
  Initiating recovery procedure
  - E.g. when waiting for responses to ACCEPT

  that creates a singleton group
  - No need for communication

The Invitation Algorithm: Example

- Consider a group of 6 nodes with ids from 1 to 6, with node 1 as leader
- Let node 1 fail
- Each of the other members forms a singleton group
- Assume that nodes 3 and 4, send invitations to the other nodes and that the conditions on the system are such that:
  - Node 2 accepts the invitation of node 3, leading to one group coordinated by node 3 and members 2 and 3;
  - Nodes 5 and 6 accept the invitation of node 4, leading to one group of coordinated by node 4 and members 5 and 6;
- Some time later, one of the nodes invites the other coordinator to join it in a group
- For an informal proof of correctness check the paper

The Invitation Algorithm: Example



- p_4 forwards the INVITATION to its group members
- To ensure liveness we need several timeouts:

  Candidate cannot wait indefinitely for:
  - accept
  - ack

  Other processes cannot wait indefinitely for:
  - ready
- Is ACK really needed?

Is the Invitation Algorithm Correct?

It appears correct

But Scott Stoller has shown that it does not satisfy Assertion 4

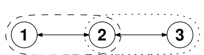
Suppose that:
1. there is a set of operating nodes R which all have two way communication with all other nodes in R. That is Assumptions 8 and 9 hold for nodes in R
2. there is no superset of R satisfying the previous property
3. no node failures occur during the election

then the election algorithm will eventually transform the nodes in set R from any state to a state where there is i in R such that for every node j

$$S(j).s = \text{NORMAL} \wedge S(j).c = i$$

...when the connectivity is not transitive

Problematic Scenario



1. Node 1 crashes
2. Nodes 2 and 3, each forms a singleton group
3. Node 1 recovers, but communication between nodes 1 and 3 has been lost. Communication between all other pairs of nodes works normally
4. Node 1 forms a singleton group, and invites node 2
5. Nodes 1 and 2 become a group, whereas node 3 becomes a singleton group.

Observation 1 If no more failures occur, these groups will not change

Observation 2 The set $\{2, 3\}$ satisfies the hypotheses on set R in Assertion 4

Contradiction
- Assertion 4 requires that node 2 be coordinator of group $\{2, 3\}$
- Node 2 is a member of group $\{1, 2\}$

Solution (1/2)

Fix the specification The specification is too strong
- It requires processes that are connected to belong to the same group
- If one of them is not the leader, that is not going to happen

Weaken the requirements But that requires a more complex definition

Two nodes are disconnected in a time interval if all messages sent between them during that interval are lost

Stable system in a time interval if, during that interval, no crashes or recoveries occur and every pair of nodes is either connected or disconnected

Connectivity graph, when a system is stable is the undirected graph whose vertices correspond to the nodes and with an edge between vertices i and j iff nodes i and j are connected

Clique cover of a graph is a partition of that graph's nodes into cliques, i.e. fully connected subgraphs

E^* reflexive and transitive closure of relation E

Solution (2/2)

Let $\langle V, E\rangle$ be the system connectivity graph

Assertion 4' For a given system, there is a constant τ such that if the system is stable for a time interval of duration at least τ, then by the end of that interval, the system reaches a state such that

a) $S(i).s =$ NORMAL $\wedge S(i).g = S(S(i).c).g \wedge (\langle i, S(i).c\rangle \in E^*)$

b) the number of groups is at most the size of a minimum-sized clique cover of $\langle V, E\rangle$

Note A clique cover is a partition of a graph's vertices in cliques. E.g. the sets

$$\{\{1, 2\}, \{3\}\}, \{\{1\}, \{2, 3\}\}, \{\{1\}, \{2\}, \{3\}\}$$

are clique covers of the problematic graph above.

Theorem The Invitation Algorithm satisfies Assertion 4'

Proof Check Scott Stoller's paper

Further Reading

- Subsection 6.5, Tanenbaum and van Steen, *Distributed Systems*, 2nd Ed.
- Hector Garcia-Molina, *Elections in a Distributed Computing System*, IEEE Transactions on Computers, Vol. C-31, No. 1, January 1982, pp. 48–59
- Scott Stoller, *Leader Election in Asynchronous Distributed Systems*, IEEE Transactions on Computers, Vol. C-59, No. 3, March 2000, pp. 283–284

Names

April 21, 2021

Outline

Server/Object Location

Problem
- Assume you want to develop a "peer-to-peer" version of the backup service on the Internet.
- How do you locate the peers storing a given chunk of a file?
  - Each file has a 256-bit id

Server/Object Location

Problem: How does a client know where is the server?

Solution: Not one, but several alternatives:
- *hard coded*, seldom;
- via program arguments: more flexible, but ...;
- via configuration file;
- via *broadcast*/*multicast*;
- via a location/name service:
  - local, e.g. *rmiregistry*.
  - *global*.

Addresses vs. Names

- Names are ... sequences of symbols (bits/characters/...) that refer to entities/objects.
- In the labs, we have used IP addresses (and ports)
- Addresses are **names** of **access points**. Or as Shoch put it:

  *The **name** of a resource indicates **what** we seek,*
  *an **address** indicates **where** it is,*
  *(and a **route** tells us **how** to get there.)*
- Addresses have some limitations:
  - Addresses often are location dependent and change frequently
    - E.g. when a service is moved from one computer to another
- Names have some advantages over addresses:
  - They can be human-friendly.
  - They can hide both complexity and dynamics
    - E.g. they can hide access point changes
- Naming is a layer of indirection
  - Ultimately you need an address to access/operate on an object

Identifiers

- An **identifier** is a name with 3 properties:
  1. an identifier refers to one entity at most;
  2. an entity has at most one identifier;
  3. an identifier refers always to the same entity (it is never reused).
- Identifiers provide a mean to refer to an entity in a precise way, independently of its access points.
- Examples?
  - From the "real" world?
  - From the "virtual" world?

Bindings, Contexts and Name Resolution

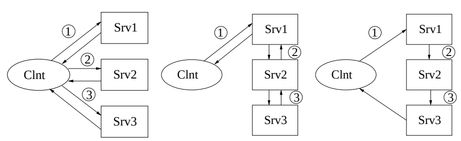
- A **binding** is a mapping from a name to an object/entity (usually identified by a lower-level name, e.g. address)
- A context/name space is a set of **bindings**
- A name space defines:
  - the syntax and structure (flat vs. hierarchical) of a name
  - the rules to find a binding of a name (**name resolution**)
- **Name resolution** is the process of finding a binding for a name
- A name is always resolved in the context of its name space:

file name	->	OS filesystem
Java program variable	->	JVM executing the program
ISBN of a publication	->	ISBN (Intern.Standard Book Number)
Car license plates	->	national/regional license plate regist

Name Resolution in a Distributed System

- Usually, name resolution is done with the help of a name service
- In small scale distributed systems, name resolution requires only one server:
  - E.g., the `rmiregistry`
- In distributed systems of larger scale, name resolution may require more than one server. In this case, name resolution can use one of 3 strategies:
  1. Iterative
  2. Recursive.
  3. Transitive.

Name Resolution: Strategies



Iterative **Recursive** **Transitive**

- Recursive name resolution:
  - Allows for caching at servers
    - This may make resolution more efficient (with lower communication costs)
  - But, it:
    - requires servers to keep state
    - makes it harder to set the values of timeouts
- Transitive name resolution also makes it harder for the client to set a timeout value

Name Resolution and *Closure Mechanism*

Names are resolved always in a context

Problem
- How do you get a context that you can use to resolve a name?
  - How do you get a "remote reference to the `rmiregistry`"?
  - How to start the name resolution of a name of a file system: i.e. where is the root directory?
  - How to find the IP address of a DNS server to resolve a DNS name?

Response

Use a **closure mechanism**
- Typically this is an *ad-hoc* and simple solution.

Hierarchical Name Spaces

- Most name spaces have a hierarchical structure:
  - OS filesystem
  - Domain Name System (DNS)
  - Postal addresses
  - Car license plates are resolved in another context – per country, region etc.
- A hierarchical structure simplifies:
  - the assignment;
  - the resolution

  of names
- Allows to partition a name space into naming domains
  - Often, a naming domain has an administrative authority for assigning names within it
  - An administrative authority may delegate name assignments for sub-domains (e.g. in DNS)

Additional Reading

- Chapter 5 of van Steen and Tanenbaum, *Distributed Systems, 3rd Ed.*
  - Section 5.1: *Names, Identifiers and Addresses*
  - Section 5.3: *Structured Naming*
- J. Saltzer, *On the Naming and Binding of Network Destinations*, in RFC 1498, 1993

Name Resolution in Flat Name Spaces

Distributed Hash Tables (DHTs)

Pedro F. Souto (`pfs@fe.up.pt`)

April 20, 2021

Resolution of Unstructured Names

Problem
- Assume you want to develop a "peer-to-peer" version of the backup service on the Internet.
- How do you locate the peers storing a given chunk of a file?
  - Each file has a 256-bit id
  - This id is **unstructured**

No solution Broadcasting/multicasting
- It just does not scale beyond a LAN

Issue How do we **resolve** efficiently an unstructured name on the Internet?

Solution Use a distributed hash table (DHT)
- Answer provided by academia to the problem of locating an entity in P2P system

Distributed Hash Table (DHT)

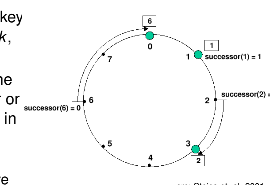
- A DHT is similar to a **hash-table**
  - It maps a **key** to a **value**
  - The **key** is an object identifier
  - The **value** is an address
    - assume it is the address of the node/peer **responsible** for the key
- The key-value pairs are stored in a potentially very large number of nodes
- A DHT provides a single operation:

  lookup(key) returns the address of the node responsible for the key
  - The address can be used to insert an object, to access to an object ...
- In a DHT-based system, node identifiers and key values are drawn from the same set, e.g. a number with m bits
- The node responsible for a key value is the one whose identifier is **closer** to that key
  - Depending on the definition of **distance** we get different DHTs

DHT Example: Chord

- Chord uses identifiers with m-bits ordered in a ring ($mod 2^m$)
- Each "object" has an m-bit random identifier: the key of DHT entries ($m = 128$ in the original paper - used `MD5`)
  - Obtained by hashing the object's key
- Each node has an m-bit random identifier
  - Obtained, e.g., by hashing the node's IP address
- The node **responsible** for key k is the **successor** of key k, $succ(k)$:

  $succ(k)$ is the node with the **smallest** id that is larger or equal to k ($succ(k) \geq k$, in modular arithmetic)
  - Given a key k the node responsible for it will have an id **higher or equal** to k.



src: Stoica et. al. 2001

Key Resolution in Chord (1/2)

Problem Given a key k, how do you find $succ(k)$?

No Solution 1 Each node n keeps information about its **successor**, i.e. the next node in the ring ($succ(n+1)$)

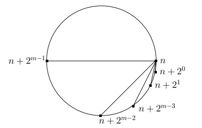
- Simple solution
- ... but it does not scale. Why?

No Solution 2 Each node n keeps information about all nodes in the ring

- Constant time name resolution
- ... but it does not scale. Why?

Key Resolution in Chord (2/2)

Solution In addition to a pointer to the next node in the ring each node keeps pointers that allow it to reduce at least in half the **distance** to the key



- Because nodes that are 2^i apart may not be active, each node n keeps a pointer to the $succ(n+2^i)$ for $i=0\ldots m-1$
- This scheme has 3 important properties:
  1. Each node keeps information on only m nodes
  2. Each node knows more about nodes closer to it than about nodes further away
  3. The table in a node may not have information on the $succ(k)$, for some k – i.e. a node may be unable to resolve a key by itself
- Key resolution requires $O(log(N))$ steps, where N is the number of nodes in the system

Chord: *Finger Table* (1/2)

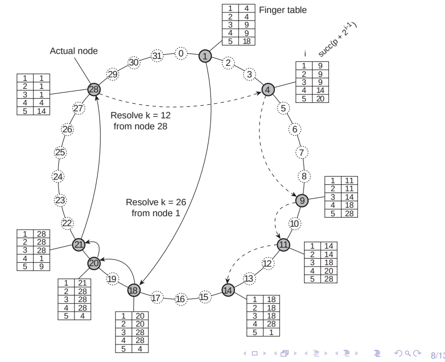
- The ***Finger table***, $FT_n[]$, is an array with m pointers:

  $FT_n[i] = succ(n+2^{i-1}) mod 2^m$ where $i = 1\ldots m$
  - $FT_n[1]$ is n's successor in the Chord ring
- To resolve (*lookup*) a key k, node n forwards the request to:
  - The next node, i.e. $FT_n[1]$, if $n < k \leq FT_n[1]$
  - To node $n' = FT_n[j]$, where j is the largest index st. $FT[j] < k$

  (All arithmetic in modulo 2^m)

  Algorithmically, n' can be computed by:
  1. Traversing the FT from the last to the first element
  2. Stopping at the element $FT_n[j]$ st: $n < FT_n[j] < k$
- Each element of the FT includes not only the node identifier but also its IP address (and port)
- Chord works correctly iff $FT_n[1]$ is correct
  - Chord tolerates transient inconsistencies in other elements of $FT_n[]$, by trying the resolution again (may not be necessary even)

Chord: *Finger Table* (2/2)



Chord: *Finger Table* (3/3)

Finger table of node 21

i	$2^{(i-1)}$	$succ(21+2^{(i-1)})$
1	1	succ(21 + 1) = 28
2	2	succ(21 + 2) = 28
3	4	succ(21 + 4) = 28
4	8	succ(21 + 8) = 1
5	16	succ((21 + 16) mod 32) = succ(5) = 9

Resolution Start at the last element of the FT, and move up until:

either FT entry is smaller than key being resolved;
or reached the first element

- If first element is larger than key, then it is its owner

@ node 1

1	4	
2	4	
3	9	
4	9	
5	18	< 26

@ node 18

1	20	
2	20	< 26
3	28	
4	28	
5	4	

@ node 20

1	21	< 26
2	28	
3	28	
4	28	
5	4	

@ node 21

1	28
2	28
3	28
4	1
5	9

Chord: Other Issues

Node Joining Node n can ask any node to locate $succ(n)$

- The crux is to get the $FT_x[1]$ correct
- Every node needs also to keep information about its **predecessor**
- Periodically:
  1. A node queries its successor about its predecessor, p
     - If p is between itself an successor
     - Then update successor to p, and notify p (new successor)
  2. Updates the elements of its FT, one at a time
  3. Checks if its predecessor is still in the ring

Node Failure Rather than keep a single successor, a node keeps a list of r successors

- If the successor fails, a node can replace it with next one

Identifiers Generation To achieve some tolerance to denial-of-service (DoS) attacks, identifiers should be generated using a cryptographic hash function, e.g. SHA256

Virtual Topology Issues (1/2)

Problem Chord, and other P2P systems, use an overlay network

- If the topology of the overlay network is oblivious to the underlying physical network, routing of messages along the overlay network may be inefficient
  - Messages may follow an erratic route, e.g. bouncing between hosts in different continents

Sol. 1: Assign identifiers according to the underlying topology

- I.e. assign identifiers so that the overlay topology is close to that of the underlying physical topology.
- This is not always possible. E.g. it is **not** possible in Chord.

Virtual Topology Issues (2/2)

Sol. 2: Route messages according to the underlying topology

- For example, Chord could keep several nodes per interval $[n+2^{i-1}, n+2^i]$ rather than a single one, and when resolving a key, might use the closest node

Sol. 3: Pick neighbors according to the underlying topology

- In some algorithms, nodes can pick their neighbors, i.e. establish the links of the overlay network.
- This is not always possible. E.g. it is **not** possible in Chord.

Further Reading

- Subsection 5.2.3, Tanenbaum and van Steen, *Distributed Systems*, 2nd Ed.
- I. Stoica et al., "Chord: A scalable peer-to-peer lookup protocol for Internet applications", IEEE/ACM Transactions on Networks, (11)1:17-32, Feb 2003 (acessível via biblioteca digital da ACM "dentro da FEUP")

Fault Tolerance

Atomic Commitment

Pedro F. Souto (pfs@fe.up.pt)

May 12, 2021

Atomic Commitment: Informal

Problem How to ensure that a set of operations executed in different processors?

- either are **all** executed (**all committed**)
- or **none** of them is executed (**all aborted**)

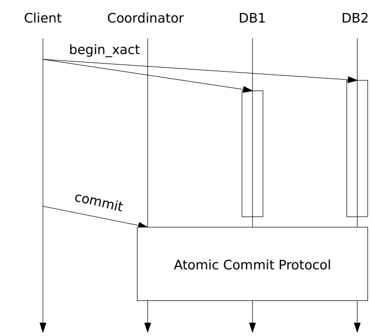
Observation The origin of this problem is distributed databases, i.e. distributed transactions:

- A transaction comprises operations (sub-transactions) in different DBs
- A transaction must be **atomic** (and also CID)

Observation AC is useful:

- Not only when processes may fail
- But also when the operations may not be performed because of some reason other than the failure of the process that execute them
  - E.g., because of a deadlock in one of the sub-transactions

Distributed transactions and Atomic Commit



Transaction's ACID[1] Properties (Reminder)

A tomicity: either all operations of a transaction are executed or none of them

C onsistency: a transaction transforms a consistent state into another consistent state

I solation: the effects of a transaction are as if no other transactions executed concurrently

D urability: the effects of a transaction that commits are permanent

[1]Coined by T. Häerder and A. Reuter in 1983

Transaction's Consistency Contract (R. Guerraoui)

System

- Atomicity
- Isolation
- Durability

Programmer

- Consistency (local)

Consistency (global)

Atomic Commitment: Formal

Def. Consider a set of n processes such that:

1. Each process has to decide one of two values: **commit**/**abort**
2. Each process shall vote/propose one of these two values
3. The value decided by each process must satisfy the following assertions:

   AC1 All processes that decide, must decide the same value

   AC2 The decision of a process is final (it cannot be changed)

   AC3 If some process decides **commit**, then all processes must have voted commit

   AC4 If all processes voted **commit** and there are no failures, then all processes must decide commit

   AC5 For any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes eventually reach a decision

Two-Phase Commit: A solution for AC (1/10)

Assumptions Processes may fail by crashing and recover

- Each process has storage whose content survives a crash

Outline The protocol has two kinds of processors:

Coordinator there is only one coordinator process, at any time instant

Participant every process that performs an operation is a participant

- The coordinator may also be a participant, in which case it will have to perform both the coordinator-side and the participant-side of the protocol

Two-Phase Commit: Basic Protocol (2/10)

Outline The protocol has two phases:

Phase 1 Upon request from the application:

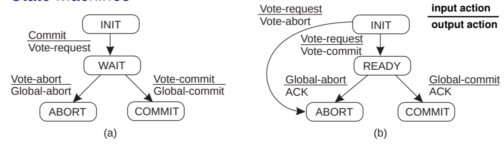
- **Coordinator** sends a VOTE-REQUEST to each participant and waits for their reply
- **Participant** upon receiving a VOTE-REQUEST each process sends its vote, either VOTE-COMMIT/YES or VOTE-ABORT/NO

Phase 2 Once the coordinator determines that is time to decide

- **Coordinator** decides/sends:
  - GLOBAL-COMMIT if it received a VOTE-COMMIT/YES from all participants
  - GLOBAL-ABORT otherwise
- **Participant** decides according to the message received from the coordinator

Two-Phase Commit: Simplified State Machine (3/10)

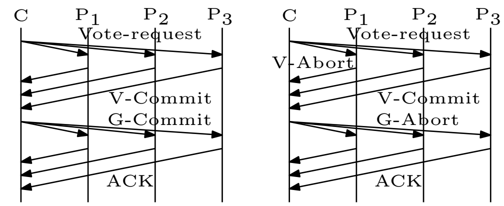
State Machines



Coordinator Participant

Two-Phase Commit: Fault-free Execution (4/10)

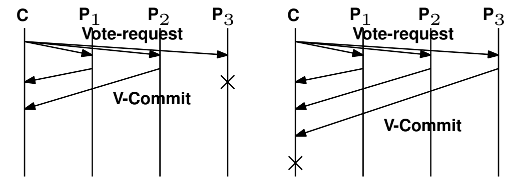
Possible Executions



- In the absence of faults, the protocol is straightforward

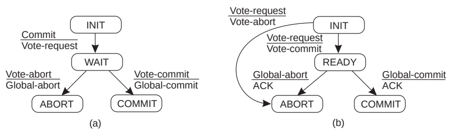
Two-Phase Commit: Faulty Executions (5/10)

Possible Executions with Faults



- We need to specify what to do in the case of failure
  - In practice, we use **timeouts** to detect failure

Two-Phase Commit: Timeouts (6/10)



Timeout Actions Actions taken by a process upon a timeout

Coordinator Waits for messages only in the WAIT state

- Decides ABORT and sends a GLOBAL-ABORT to participants

Participants Waits for messages both in:

- INIT Decide ABORT and move to corresponding state
- READY Must execute a **termination protocol** to find out the outcome.

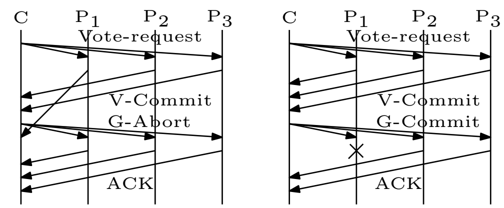
Two-Phase Commit: Termination Protocol (7/10)

Participant must communicate with the other participants to find out the outcome

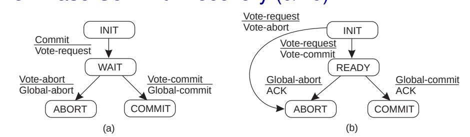
- If some participant has voted VOTE-ABORT ...
- If some participant knows the decision ...
- Otherwise
  - Process must wait until it learns the coordinator's decision
  - May continue probing both the coordinator and other participants

Two-Phase Commit: Execution with Faults (8/10)

Possible Executions with Faults



Two-Phase Commit: Recovery (9/10)



Recovery Actions Actions taken by a process upon recovery from a crash

- Assumes that each process keeps in stable storage the state of the 2PC protocol
- If process has not decided yet then
  - If crashed while waiting for a message, take the corresponding timeout action
    - Including the execution of a termination protocol, if a participant failed in the READY state
  - Otherwise (coordinator in the INIT state) decide ABORT

Two-Phase Commit: Recovery (10/10)

- To allow recovery, processes must write the state of the protocol as entries to a **log in stable storage**
  - The write of the entry should be performed before or after sending the corresponding messages?
  - In addition to the state of the protocol, the log may be used to store application data
- The 2-phase commit protocol satisfies assertions AC1-AC5, even in the presence:
  - non-byzantine node failures
  - communication faults, including partitions
- The main problem with this protocol is that it may require **participants** to block (wait longer than a communication timeout)
  - This problem can be made less likely by using the three-phase commit protocol

Atomic Commit: Independent Recovery and Blocking

Impossibility of independent recovery there is no AC protocol that always allows local recovery, i.e. without communication with other processes

- If a process is uncertain when it fails, ...

Non-blocking impossibility there is no AC protocol that never blocks in the presence of either:

Communication failures

- If a participant becomes isolated when it is in uncertain state ...

Failures on all other processes

2 Phase-Commit may block even when there is no failure on all sites:

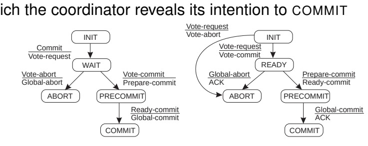
Can we do better?

Assuming:

1. Communication is reliable
2. Process failure can be reliably detected

Three-Phase Commit (1/2)

- Adds a phase between the 2 phases of the 2PC protocol, in which the coordinator reveals its intention to COMMIT



Coordinator Participant

- The PRECOMMIT states ensure the **non-blocking** condition:
  - No process can commit while another process is in an uncertain state (INIT, WAIT, READY), i.e. can decide either way
  - **Note** a process in the PRE-COMMIT state is not uncertain:
    - It will decide **commit**, unless the coordinator fails
- It can be shown that this condition is necessary and sufficient to prevent blocking **unless ...**

Three-Phase Commit (2/2)

All processes fail

There is no majority

- If a majority is able to communicate:
  - If a majority of the participants are in the READY state, they can decide ABORT
  - If a majority of the participants are in the PRECOMMIT state, they can decide PRECOMMIT
- Because two majorities must overlap, no different decisions can be made
  - Note that a process in the PRECOMMIT state may still ABORT, but only if the coordinator fails
- If there is no majority (as a result of network partition, e.g.) processes may block

Need appropriate timeout/termination/recovery actions

- Anyway, virtually all systems implement 2PC, not 3PC

What if communication is not reliable?

- I.e., communication failures are not masked by retransmissions?

Stable Storage (1/2)

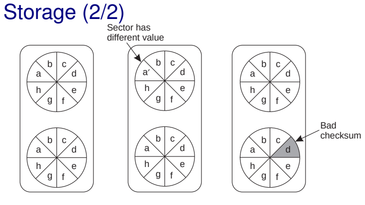
Problem

- Many protocols like 2PC assume that the state of processes, or at least some part of it, survives the failure of the process
- Usually, this means to store the state on disk
- How do you ensure that the data survives disk failures?

Solution: Stable Storage

- Use two identical disks
- Writing a block requires writing first in disk 1 and then in disk 2
- Upon reading a block, try disk 1 first, unless its **checksum** is not valid

Stable Storage (2/2)



Recovery after a crash

- If the checksum of disk 1 is valid, and the two blocks are different, copy block from disk 1 to disk 2
- If the checksum of disk 1 is not valid, use the block on disk 2, if its checksum is valid
- If the checksums of both disks are not valid, then **data has been lost**, i.e. we have a **catastrophic-failure**

Further Reading

- Chapter 8, Tanenbaum and van Steen, *Distributed Systems*, 2nd Ed.
  - Section 8.5: Distributed Commit
  - Paragraph 8.6.1: Stable Storage
- P. Bernstein, V. Hadzilacos and N. Goodman, *Distributed Recovery*, Chp. 7 of *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987

Hierarchical Name Services: DNS

April 21, 2021

Roadmap

- The Problem
- DNS: A Hierarchical Name Service
- Resource Records
- Name Resolution
- Replication in DNS
- Further Reading

DNS Names and IP Addresses

Problem: IP addresses are not easy to memorize (even in v4, let alone v6);

Solution: use **names** instead of addresses:
- identify objects;
- (may) help locate objects;
- (may) specify a role;
- (may) provide access permissions

Roadmap

- The Problem
- DNS: A Hierarchical Name Service
- Resource Records
- Name Resolution
- Replication in DNS
- Further Reading

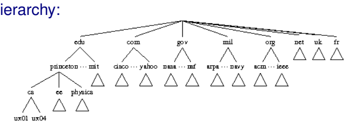
DNS: Domain Name System

- Is used on the Internet to identify objects (not only hosts):
  - DNS uses a hierarchical name space;
  - This space is maintained in a distributed and hierarchical way by several servers
- Before DNS, the Internet used a file that was maintained by the Internet Network Information Center (NIC), and initially distributed periodically to all computers. This was a:
  - centralized
  - non scalable

  solution

DNS: Fundamental Concepts

Hierarchy:

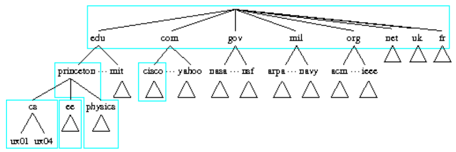


Name: `up.pt` or `fe.up.pt` (starts at the node);

Domain (subdomain): sub-tree under a name: `up.pt` (`fe.up.pt`).

Name Space Implementation: Zones

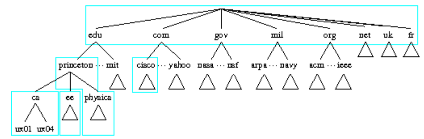
Zone: is a sub-tree that stems from the partition of the DNS hierarchy
- A zone may not be a subdomain. E.g. the `princeton.edu` zone



- 2 zones never overlap
- A zone corresponds to an administrative authority

DNS Server

- For each zone there is one server, the primary, that is responsible for the information of that zone



- A server may contain information of more than one zone.
- For availability reasons, a zone's information must be replicated at least in another (secondary) server

IMP- Primary and secondary servers should be located so as to maximize the availability of the zone's information

Roadmap

- The Problem
- DNS: A Hierarchical Name Service
- Resource Records
- Name Resolution
- Replication in DNS
- Further Reading

Resource Records

- The information on each node in the DNS tree is kept in resource records
- A resource record maps the node's name to a value:
  (name, value, type, class, ttl)

  where

  `name/value` are not necessarily host names and IP addresses

  `type` specifies how the value should be interpreted (it depends on the class);

  `class` specifies the name space (the `IN` class is, by far, the most used class);

  `ttl` *time-to-live* - is the validity time of the record in seconds (used for caching)

Resource Records: `IN` Class Types

`A` IP address (`dig sifeup.fe.up.pt`)
(sifeup.fe.up.pt, 193.136.28.205, A, IN)

`NS` name of the zone's DNS server zona (`dig ns fe.up.pt`)
(fe.up.pt, ns1.up.pt, NS, IN)

`CNAME` alias (canonical name) (`dig cname www.fe.up.pt`):
(www, sifeup.fe.up.pt, CNAME, IN)

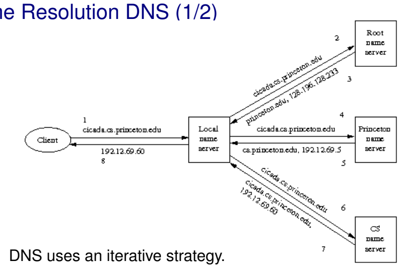
Cannonical names do not have type A RR

`MX` name of SMTP servers (`dig mx fe.up.pt`):
(fe.up.pt, mx01.up.pt, MX, IN)
...
(fe.up.pt ,mx06.up.pt, MX, IN)

Roadmap

- The Problem
- DNS: A Hierarchical Name Service
- Resource Records
- Name Resolution
- Replication in DNS
- Further Reading

Name Resolution DNS (1/2)



- DNS uses an iterative strategy.
- Resolution is based on **longest prefix matching**
- Every server must keep `(NS,A)` RR pairs for each of its direct subdomains
- Clients must be configured with either a local name server or a public name server, e.g. Google's

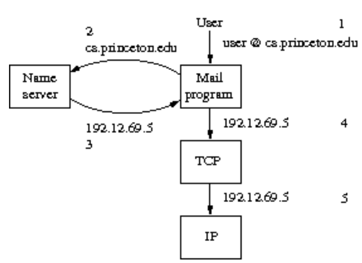
Resolução de nomes em DNS (2/2)

- For efficiency reasons, i.e. avoiding communication, DNS **resolvers**, i.e. DNS clients, use caching extensively

  Libraries not as effective, but may avoid communication

  Servers , a.k.a. **recursive resolvers**
- Answers provided using caches are said to be **non-authoritative**
- Cached RRs are garbage-collected using their TTL field.

Exemplo: Uso de DNS



Obs.- The client needs both the MX RR for `cs.princeton.edu` and the A RR of the SMTP server. However, it enough to send an MX query.

Roadmap

- The Problem
- DNS: A Hierarchical Name Service
- Resource Records
- Name Resolution
- Replication in DNS
- Further Reading

Replication in DNS (1/2)

- DNS relies heavily on replication for reasons of:
  - performance
  - availability
- Replication raises problems of inconsistency
  - Which are even harder when changes can be done on different copies
- DNS has some special features that simplify replication
  - The zone information is updated/added at only one server (the primary)
  - The replicas need not be updated synchronously, i.e. simultaneously. The use of stale data
    - Is usually detected upon use
    - Even if that is not the case, it usually does not affect the correction of the applications that use DNS

Replication in DNS (2/2)

- In DNS a zone's RRs must have
  - One **primary server**
  - At least one **secondary server**
- Both primary and secondary servers must keep all RRs of the nodes in the zone
  - Replies by these servers are considered **authoritative**
- To detect changes to a zone's RR, each zone has a Start of Authority (SOA) RR with the following fields

Serial a 32 bit that identifies the zone's "version"
- Everytime there is a change to a Zone's RR, this field must be increased

Refresh a 32 bit integer that specifies the maximum time (in seconds) between update attempts

Update Detection and Zone Propagation

Update detection by comparing the Serial field of the SOA RR in the secondary with that of the SOA RR in the primary

How can the secondary get the primary's SOA RR?

polling the secondary issues a SOA query to the primary

notification every time the primary updates the serial number of its SOA, it notifies the secondary servers
- This is specified in RFC 1996, but it has not been approved yet

Zone Transfer

Non-incremental the secondary sends an `AXFR` query requesting for the transfer of an entire zone

Incremental the secondary requests the primary to transfer only the changes to the zone.
- This approach is specified in RFC 195, but it has not been approved yet

Note DNS requires zone transfer to use TCP
- For all other queries, DNS usually uses UDP, although it is possible to use TCP

Roadmap

Further Reading

- Subsection 5.3(.4) of van Steen and Tanenbaum, *Distributed Systems, 3rd Ed.*
- P. Mockapetris, *DOMAIN NAMES - CONCEPTS AND FACILITIES*, in RFC 1034, 1987
- P. Mockapetris, *DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION*, in RFC 1035, 1987
- M. Ohta, *Incremental Zone Transfer in DNS*, in RFC 1995, 1996
- P. Vixie, *A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)*, in RFC 1996, 1996

Clock Synchronization

Pedro F. Souto (`pfs@fe.up.pt`)

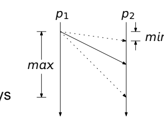
April 28, 2021

Roadmap

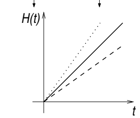
Roadmap

Distributed System Model

- A set of sequential processes that execute the **steps of a distributed algorithm**
  - DS are inherently concurrent, with real parallelism
- Processes communicate and synchronize by exchanging messages
  - The communication is not instantaneous, but suffers delays



- Processes may have access to a local clock
  - But local clocks may drift wrt real time



- DS may have partial failure modes
  - Some components may fail while others may continue to operate correctly

Fundamental Models

Synchronism characterizes the system according to the temporal behavior of its components:
- processes
- local clocks
- communication channels

Failure characterizes the system according to the types of failures its components may exhibit

Models of Synchronism

Synchronous iff:
1. there are known bounds on the time a process takes to execute a step
2. there are known bounds on the time drift of the local clocks
3. there are known bounds on message delays

Asynchronous No assumptions are made regarding the temporal behavior of a distributed system
- These 2 models are the extremes of a range of models of synchronism

Dilemma
- It is relatively simple to solve problems in the synchronous model, but these systems are very hard to build

Roadmap

Clock Synchronization

Observation
- In our every-day lives we use time to coordinate our activities
  - Distributed applications can do the same, as long as each process has access to a "sufficiently" **synchronized clock**

Synchronized Clocks
- Are essential for implementing:

some basic services:

Synchronous distributed algorithms i.e. rounds

Communication protocols e.g. TDMA

some distributed applications:

Data collection and event correlation

Control systems

GPS

- Enable to reduce communication and therefore improve performance (Liskov 93)

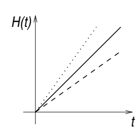
Applications (according to David Mills)

- Distributed database transaction journaling and logging
- Stock market buy and sell orders
- Secure document timestamps
- Aviation traffic control and position reporting
- Radio and TV programming launch
- Real-time teleconferencing
- Network monitoring, measurement and control
- Distributed network gaming and training
- Take it with a grain of salt

src: David Mills, Network Time Protocol (NTP) General Overview

Local Clocks

- Computers have hardware clocks based on quartz crystal oscillators, which provide a local measure of time, $H(t)$. Often:
  - These clocks generate periodic interrupts
  - The OS uses these interrupts to keep the local time



- These clocks have a **drift** $\left(\frac{dH(t)}{dt} - 1\right)$ wrt a perfect clock
  - Quartz-based oscillators have drifts in the order of 10^{-6}, i.e. they may run faster/slower a few μs per second
  - The clock drift of a quartz-based oscilator depends on the environmental conditions, especially temperature
- Even if this drift is **bounded**, unless clocks **synchronize**, their values may diverge forever, i.e. their **offset/skew** $(H_i(t) - H_j(t))$, may grow unbounded

What is Clock Synchronization?

External Clock Synchronization (CS)

I.e. synchronization wrt an external time reference (R)

$$|C_i(t) - R(t)| < \alpha, \forall i$$

where α is known as the CS **accuracy**

Internal Clock Synchronization

I.e. synchronization among the local clocks in the system – needs not be related to **real time**

$$|C_i(t) - C_j(t)| < \pi, \forall i, j$$

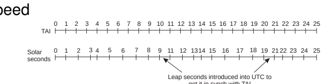
where π is known as the CS **precision**

Note that external clock synchronization implies internal clock synchronization.
- What's the precision?

Time Scales/Standards

International Atomic Time (TAI) This is a weighted average of the time kept by around 300 atomic clocks in over 50 national labs
- More stable clocks are given larger weights
- It uses GPS satellites for clock comparisons that provide the data for the calculation of TAI

Coordinated Universal Time (UTC) It is derived from the TAI, by adding leap seconds to compensate for variations in Earth's rotation speed



GPS provides time with an accuracy better than 100 nanoseconds
- Each satellite has its own atomic clocks (3 or 4), which are kept synchronized with the Master Clock at US Naval Observatory
- The MC@USNO is a set of more than 40 atomic clocks that produces UTC(USNO), which is used for computing the UTC

Roadmap

Clock Synchronization: Centralized Algorithm

Assumptions

Each process has a local clock

- Master/server clock provides the time reference
- Slave/client clock synchronize with the master/server clock

Local clock drifts are bounded

$$(1+\rho)^{-1} \leq dC_i(t)/dt \leq 1+\rho, \text{for all correct processes } i$$

System is synchronous

- There are known bounds (both lower and upper) for communication delays
- The time a process requires to take a step is negligible with respect to the communication delays (and therefore bounded)

Clock Synchronization: Specification

Accuracy

$$|C_i(t) - C_m(t)| \leq \alpha, \text{for all correct processes } i$$
$$\text{where } C_m(t) \text{ is the master's clock}$$

Centralized Clock Synchronization: Idea

Master

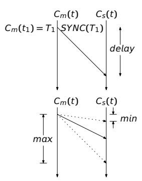
- Periodically
  - read the local time
  - broadcast it in a SYNC message

Slave

- Upon reception of a SYNC message,
  - **update** the local clock

Master Time Estimation

Issue There is a *delay* between the reading of the time @ the master, and its processing @ the slaves. By the time the slave gets the message, the time @ the master will be:



$$C_s(t) = C_m(t) + delay$$

Problem What is the value of *delay*?

Response Do not know, but can be estimated:

$$min < delay < max$$

max is known because the system is synchronous

- To minimize the error, should use $[C_m(t) + min, C_m(t) + max]$ midpoint:

$$C_s(t) = C_m(t) + \frac{min + max}{2}$$

SYNC Message Period

To ensure **accuracy**, it must be;

$$|C_m(t) - C_s(t)| < \alpha$$

Because the clock drifts are bounded:

$$\left|\frac{dC_s(t)}{dt} - \frac{dC_m(t)}{dt}\right| < 2\rho$$

Hence,

$$\frac{max - min}{2} + 2\rho T < \alpha$$

Observation The clock skew lower bound is determined by the **delay jitter** ($(max - min)$)

- If T is large, the second term may dominate
  - NTP sometimes uses a T of 15 days, and $\rho \sim 10^{-6}$

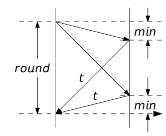
Message Delay Estimation in Practical Systems (1/2)

Issue Often, there is no known upper bound for the communications delay

Approach (Christian) Estimate it based on the round-trip-delay, *round*

Let *min* be the lower delay bound
Then, when the slave receives the master's clock reading, time at the master will be in the interval:

$$[t + min, t + round - min]$$



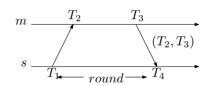
By choosing the **midpoint** in this interval, i.e. by assuming that the delay is $\frac{round}{2}$, we **bound the error** to:

$$\frac{round}{2} - min$$

Message Delay Estimation In Practical Systems (2/2)

Insight By measuring the time the server takes to reply, we can reduce the error

- This can be implemented by timestamping with the **local clocks** the sending/receiving of messages



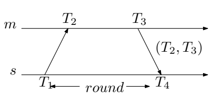
Let $C_s(t) = C_m(t) + O(t)$
Then,
$T_2 = T_1 - O(t_1) + d_1$
$T_4 = T_3 + O(t_3) + d_2$

Note: Upper-case T's are **clock times**, lower case are **real-times**

Assume

- Synchronized rates $O = O(t_1) = O(t_3)$
- Symmetry $del = (d_1 + d_2)/2 = ((T_4 - T1) - (T3 - T2))/2$

Clock Offset Estimation Using Timestamps



Let $C_s(t) = C_m(t) + O(t)$
Then,
$T_2 = T_1 - O(t_1) + d_1$
$T_4 = T_3 + O(t_3) + d_2$

Note: Upper-case T's are **clock times**, lower case are **real-times**

Assume $O = O(t_1) = O(t_3)$
Then $O = \tilde{O} + (d_1 - d_2)/2$
Where $\tilde{O} = ((T_4 - T_3) + (T_1 - T_2))/2$
Remember: $del = (d_1 + d_2)/2 = ((T_4 - T1) - (T3 - T2))/2$
Because $d_1, d_2 \geq 0$, then $d_1 - d_2 \leq d_1 + d_2$
Therefore: $\tilde{O} - del \leq O \leq \tilde{O} + del$
$\tilde{O}$ can be seen as an estimate of the offset
del can be seen as an estimate of $\tilde{O}$'s error bound

Clock Offset and Delay Time Estimation Precision

Precision depends on:

- Rate synchronization i.e. $O(t_1) = O(t_3)$
- Symmetry assumption i.e. $d_1 = d_2$, only in the case of **delay estimation**
- Timestamp accuracy
  - The lower in the protocol stack they are generated the better
  - In some cases, the timestamps are sent in later messages, PTP calls them FOLLOW-UP messages.

Precision Time Protocol (PTP - IEEE 1588)

Description Protocol for clock synchronization with high precision on packet-based networks

Goals

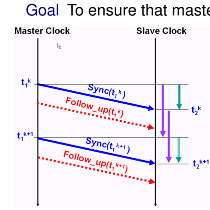
- Precision of at least one microsecond
- Minimum hardware resource requirements
- Minimum administration on single subnet systems
- Applicable, but not limited, to Ethernet

Application domains

- Test and measurement
- Industrial automation
- Power industry
- Telecoms
- Aerospace, navigation and positioning
- Aujdio and video networks

Precision Time Protocol: Syntonization

Goal To ensure that master and slaves have the same clock rate

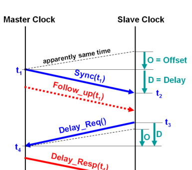


- Two alternative ways to send the T_i^k timestamp
  - **Sync** message: requires the ability to insert the timestamp into the message on the fly
    - This is an issue if the timestamps are generated in the lower layers (see below)
  - **Follow_up** message: requires an additional message
- Slave clock rate is adjusted so that:

$$T_1^{k+1} - T_1^k = T_2^{k+1} - T_2^k$$
$$T_2^{k+1} - T_1^{k+1} = T_2^k - T_1^k$$

- Sync messages are sent periodically with a period of a few seconds
  - To correct variations due to changing environmental conditions, e.g. temperature

Precision Time Protocol: Offset and Delay Estimation



- **Sync** message (possibly with Follow_up) is used to obtain timestamps T_1 and T_2
- **Delay_Req** and Delay_Resp messages are used to obtain timestamps T_3 and T_4

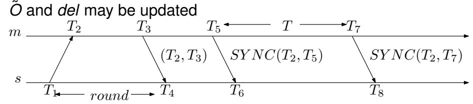
- Slave clock rate is adjusted so that:

$$T_1^{k+1} - T_1^k = T_2^{k+1} - T_2^k$$
$$T_2^{k+1} - T_1^{k+1} = T_2^k - T_1^k$$

- Sync messages are sent periodically with a period of a few seconds
  - To correct variations due to changing environmental conditions, e.g. temperature

Precision Time Protocol

- By including the appropriate timestamps in the SYNC messages $\tilde{O}$ and *del* may be updated



- $\tilde{O}$ can be used later to estimate the value of the master clock at the **time of reception** of a SYNC message with the master's time:

$$Time(SYNC) - \tilde{O}$$

Local Clock Correction

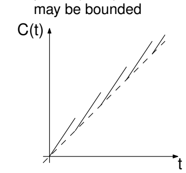
Instantaneous correct offset only at every synchronization point

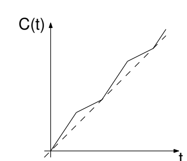
- May lead to non-monotonic clocks

Amortized over the synchronization period, i.e. adjust both *a* and *b*

$$C_s(t) = at + b$$

- So as to ensure that:
  - Local time is continuous
  - Minimize the error at the end of the synchronization period, possibly subject to the constraint that the change in the rate may be bounded





IMP: Each clock between resynchronization points can be

Roadmap

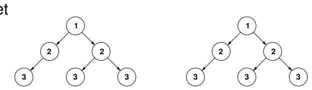
Network Time Protocol (NTP)

- Clock synchronization protocol designed for the Internet
  - tolerates communication delay jitter
  - it is robust against loss of connectivity
  - scales to a large number of clients
  - it is robust against interferences, whether malicious or accidental

Mills, *Internet time synchronization: the Network Time Pro tocol*, IEEE Trans. on Communications, October 1991, 1482-93.
Mills, *Improved Algorithms for Synchronizing Computer Network Clocks*, IEEE Trans. on Networks, June 1995, pp. 245-54.

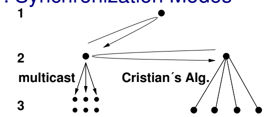
NTP: Architecture

- NTP servers are organized hierarchically in a synchronization subnet



- Servers at the top (**primary**) are connected to a UTC source
- Secondary servers synchronize with the primary and so on
- The leaves of the trees are clients
- Each level of the hierarchy is known as a **stratum**, the lowest stratum comprises the primary servers
- The synchronization subnet may reconfigure in the sequence of failures. E.g.:
  - If a UTC source becomes unreachable, a primary server may become secondary
  - If a primary server becomes unreachable, a secondary server may switch to another primary server

NTP: Synchronization Modes



Multicast a server multicasts periodically its time to clients
- The precision is relatively low, but is OK for LANs (why?)

Request-reply similar to Cristian's algorithm.

Symmetrical in which servers swap their times. Used by servers at the lowest strata. (Ensures the highest precision.)
- I.e. at this level, NTP uses a **decentralized** algorithm

- Besides that, NTP uses lots of statistics to compensate for the the jitter in communication delay
  - It is this variability that limits the precision
  - The OS itself also introduces some jitter
- NTP uses UDP always

Roadmap

Synchronized Clocks: Practical Usage

- On the Internet, it is possible to have synchronized clocks with:
  - A small **enough skew** (δ)
  - A high **enough** probability
- We can take advantage of this to reduce:
  - communication
  - state (stored in stable storage)

  and therefore improve performance
- What if clocks get ouf of sync? Depends:

  Compensate at a higher level
  Do nothing there are different reasons:
  1. Correctness is not affected
  2. Domination of other failures
  3. Engineering trade-off: performance benefits outweight failure costs

Synchronized Clocks: At-most-once Messaging (1/3)

Simplistic solution
- Remember all the messages received
- Discard duplicated messages

Issue: Cannot remember ALL messages received

Solution Forget **far away past**

Session-based
- Node has to execute some handshake before sending the first message after a while
  - In response, the receiver generates a **conn(ection) id**;
  - The sender must tag subsequent messages with the conn id
  - The handshake may be too high overhead

Synchronized clocks
- All nodes have a synchronized clocks with accuracy α

Synchronized Clocks: At-most-once Messaging (2/3)

Each message has:

A connection id
- selected by the sender - no need for handshake
- must be unique (also among all senders)

A timestamp

Receivers keep:

A connection table (CT) with, for each connection:
- The timestamp, *ts*, of the last message received
  - If not replaced by that of more recent message, it is kept at least for the life-time, λ, of a message in the network, i.e. as long as $ts > time - \lambda - \alpha$ (Why subtracting α?)

An upper bound, *upper*, on the value of all timestamps removed from the connection table.

Upon reception of a message it is discarded if its timestamp is:
- either smaller than that of the entry of the connection table with the corresponding connection id
- or smaller than *upper* (if there is no entry in the CT)

Synchronized Clocks: At-most-once Messaging (3/3)

Issue: What if the receiver recovers after a crash?
- Must ensure that messages delivered before the crash are not delivered again

Solution

Naïve store in **stable storage** the largest timestamp of all messages received, and discard any message with a smaller ...

Efficient Periodically save in stable storage an upper-bound, $latest = time + \beta$, of the time-stamp of all delivered timestamps
1. Upon reception of a message with a timestamp larger than *latest* either delay its delivery or discard it
2. Upon recovery from a crash:
   - Set *upper* to *latest*, thus discarding messages whose timestamps are older than the saved upper-bound.

Trade-off
- Synchronized clocks improve performance
- At the cost of occasionally rejecting a message

Synchronized Clocks: Leases

Originally were just like a **timed locks**
- Locks are difficult to manage in an environment where lock owners may crash
- It is even possible to have read/write leases

Generally They can be used to ensure some property during a limited time, i.e. the **lease duration**

Lease duration may be

Absolute requires **synchronized clocks**
Relative requires **synchronized rates**

Renewals A lease is valid for its duration
- Unless its owner **renews** the lease

Question How long should the lease duration be?

Synchronized Clocks: Preventing Replay-Attacks

Kerberos uses Needham and Schroeder's shared key authentication protocol
- But instead of using their solution to prevent replay attacks, they use synchronized clocks
  - And, more generally, to prevent the use of keys for too long, i.e. for key expiration
- Check the details in Liskov's paper

Synchronized Clocks vs. Synchronized (Clock) Rates

- In most applications, synchronized rates are enough
  - In computer networking, retransmission mechanisms usually assume clock rates synchronized with real time rates
  - TCP also assumes that to prevent segments sent in the scope of a connection to be delivered in the scope of another connection (when a connection is shutdown abruptly)
- This is a reasonable assumption for computer systems with quartz clocks
  - Nowadays, these clocks have a drift rate lower than 10^{-6}
- Whenever possible **synchronized rates** should be used rather than **synchronized** clocks
  - Synchronized clocks depend on synchronized rates, and on occasional communication
  - But the use of synchronized rates also requires communication, so that the instant of the relevant event can be bounded
- Synchronized clocks are more powerful than synchronized rates
  - They are able to support other algorithms for which synchronized rates are not enough
  - They can provide warnings when the clocks get out of synch

Roadmap

Further Reading

- Tanenbaum and van Steen, Section 6.1 of *Distributed Systems*, 2nd Ed.
- NTP Project Page
- Barbara Liskov, *Practical Uses of Synchronized Clocks*, in Distributed Computing 6(4): 211-219 (1993)
- Time at the Bureau International des Poids e Mesure, keeper of the TAI
- Time and Frequency Division of the NIST Physical Measurement Lab
- Precise time @ US Naval Observatory, owner of the Master Clock used in GPS

Synchronization

Lamport Logical Clock and Clock Vectors

Pedro F. Souto (pfs@fe.up.pt)

May 5, 2021

Roadmap

Roadmap

Events

- Often, there is no need for synchronized clocks
  - Sometimes, we just need to order **events**

Event is an action that takes place in the execution of an algorithm

Sending of a message
Delivery of a message
Computation step

Execution of a distributed algorithm by process i can be defined as a sequence of events:

$$e_i^0 e_i^1 e_i^2 \ldots$$

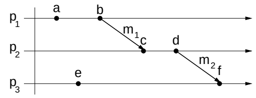
- Each event in an execution is an instance of a given **event type**

The Happened-Before Relation ($\rightarrow$)

HB1 If e and e' are events that happen in that order in the same process, then $e \rightarrow e'$

HB2 If e is the sending of a message m and e' is the reception of that message, then $e \rightarrow e'$

HB3 If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$
- That is, the HB relation is transitive



- The HB relation is a partial order. For example:
$$a \not\rightarrow e \wedge e \not\rightarrow a$$
  - Events like a and e are **concurrent**: $a || e$
- The HB relation captures the **potential causality** between events

Roadmap

The Happened-Before Relation

Lamport Clocks and Timestamps

Vector Clocks and Timestamps

Lamport Clocks and Timestamps

Lamport clock is a logical clock used to assign (Lamport) timestamps to events
- Each process in the system has its own Lamport clock L_i

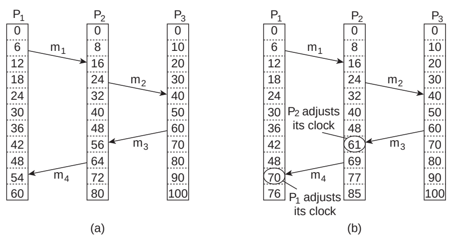
(Lamport) clock condition For any events e and e':
$$e \rightarrow e' \Rightarrow L(e) < L(e')$$
Or equivalently:
$$L(e) \geq L(e') \Rightarrow e \not\rightarrow e'$$

Satisfying the Clock Condition

C1 If e and e' are events in process p_i and occur in that order, then $L_i(e) < L_i(e')$

C2 If
e is the sending of a message by process p_i, and
e' is the receipt of that message by a process p_j
then $L_i(e) < L_j(e')$

(Physical Clocks vs. Lamport Clocks)



- Free-running physical clocks cannot be used as Lamport clocks
  - If the clock resolution is small enough C1 is easy to ensure
  - The problem is with C2

Lamport Clocks and Timestamps (2/3)

- The timestamp of an event at process i is assigned by L_i, the Lamport clock at the process i

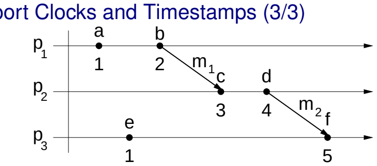
Lamport Clock Update Rules To satisfy the Clock Condition a Lamport clock must be updated **before** assigning its value to the event as follows:

LC1 if e is not the receiving of a message, just increment L_i

LC2 if e is the receiving of a message m:
$$L_i = max(L_i, TS(m)) + 1$$
where $TS(m)$ is the timestamp of the corresponding sending event
- Implementing LC2 requires piggybacking the timestamp of the sending event on every message

IMP. Incrementing a Lamport clock is **not** an event

Lamport Clocks and Timestamps (3/3)



- If we need to order all events we can use the pair (**extended Lamport timestamp**):
$$(L(e), i)$$
where i is the process where e happens
  - How would you define that order, so that it "extends" the HB relation?
  - Although total, this order is somewhat arbitrary
- Actually, Lamport claims that the reason for Lamport clocks is precisely to obtain a total order.

State Machine

- Indeed, a total order allows to solve any synchronization problem

Idea Specify the synchronization in terms of a state machine:

Set of commands, C
Set of states, S
State transition function, $t : C \times S \rightarrow S'$
- I.e., the execution of a command, c, changes the current state s to a new state s', formally: $t(c, s) = s'$

Synchronization is achieved, if all processes execute the same set of commands in the same order

A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T. The precise algorithm is straightforward, and we will not bother to describe it.

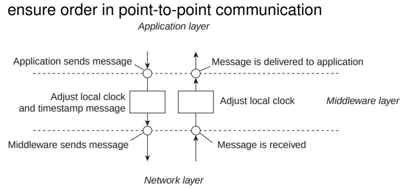
The problem is that the failure of a single process blocks the system

Use of Lamport Clocks: Total Order Multicast (1/2)

Problem How to ensure that if m and m' are delivered by processes i and j, then they deliver m and m' in the same order?

Solution Use **extended** Lamport timestamps to timestamp messages, and **deliver** the messages in the order of these timestamps

Deliver vs. Receive this is similar to what happens with TCP to ensure order in point-to-point communication



Use of Lamport Clocks: Total Order Multicast (2/2)

Assumptions The communication channel is:
Reliable
FIFO

- Each process keeps its own Lamport clock
- The only relevant events are the sending and receiving of multicast messages
- Just before multicasting a message, the LC is incremented and its value used to timestamp the message
- Upon receiving a message m, a process:
  1. Inserts the message in a queue of messages ordered by their extended Lamport timestamps;
  2. Updates the Lamport clock to $LC = max(TS(m), LC)$
- A message is delivered to the application only when:
  - It is at the head of the queue
  - It is **stable**
    - If there is a message on the queue from every other process
- To reduce the delivery delay, can use acknowledgments

Roadmap

The Happened-Before Relation

Lamport Clocks and Timestamps

Vector Clocks and Timestamps

Vector Clocks (1/3)

Observation The main limitation of Lamport clocks is that:
$$L(e) < L(e') \not\Rightarrow e \rightarrow e'$$

Idea Use an array of timestamps, one per processor
- Due to Mattern, Fidge and Schmuck

Rules Each process p_i keeps its own vector V_i, which it updates as follows:

VC1 if e is not the receiving of a message, just increment $V_i[i]$

VC2 if e is the receiving of a message m:
$$V_i[i] = V_i[i] + 1$$
$$V_i[j] = max(V_i[j], TS(m)[j])$$

- **Initially** $V_i[j] = 0$, for all j
- The timestamp of the sending event is piggybacked on the corresponding message ($TS(m) = V(send(m))$)

Basically
$V_i[i]$ is the number of events timestamped by p_i
$V_i[j]$ is the number of events in p_j that p_i **knows** about

Vector Clocks (2/3)

Let V and V' be vector timestamps

Vector Timestamps Comparison We define the relation $<$ between vector timestamps:
$$V < V' \text{ iff } \forall_j : V[j] \leq V'[j] \wedge \exists_i : V[i] < V'[i]$$

Vector Clock Condition

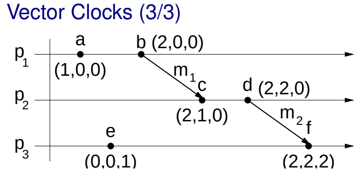
$$e \rightarrow e' \quad \Rightarrow V(e) < V(e')$$
$$V(e) < V(e') \quad \Rightarrow e \rightarrow e'$$

On the other hand:

$$\neg(V(e) < V(e')) \wedge \neg(V(e') < V(e)) \Rightarrow e || e'$$

Conclusion Vector timestamps can be used to determine whether the HB relation holds between any two pairs of events
- Lamport clocks allow to conclude **only** if the HB does **not** hold

Vector Clocks (3/3)



- a and e are concurrent events
- The main issue with vector clocks is that we need *n timestamps* per event, whereas *Lamport clocks* need only one
  - But there is no way to avoid it (Charron-Bost).
- **Hidden** communication channels can lead to **anomalous** behavior
  - I.e. to the violation of the Clock Condition

for both Lamport clocks and vector clocks.
  - Lamport claims that only synchronized (physical) clocks may eliminate such anomalies

Vector Clocks Use: Causal Order Multicast (1/2)

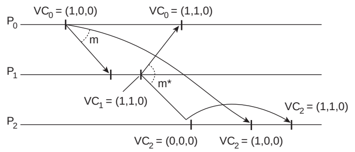
Problem How to ensure that if $m \rightarrow m'$ and process i delivers m', then it must have previously delivered m

Solution Use vector clocks to timestamp messages, and **deliver** the messages in the order of these timestamps (as defined above)
- Each process keeps its own vector clock
- The only relevant events are the sending and receiving of multicast messages
- Just before multicasting a message, the sender updates its VC after which it timestamps the message
- Upon receiving a message m, a receiver inserts the message in the queue of received messages
- Process i delivers message m to the application only when:

$$V_i[j] \geq TS(m)[j], \forall j \neq k, \text{ where } k \text{ is the sender of } m$$
$$V_i[k] = TS(m)[k] - 1$$

After which it should update its VC (no need to increment $V_i[i]$)

Vector Clocks Use: Causal Order Multicast (2/2)



Observations

- $VC_i[i]$, counts the number of messages multicasted by p_i
- $VC_i[j]$, $i \neq j$, counts the number of messages multicasted by p_j that p_i has delivered to the application
- The communication channels need to be reliable, but not FIFO. Why?

Question does the total order multicast protocol (with Lamport timestamps) also ensure causal order?

Further Reading

- Tanenbaum and van Steen, Section 6.2 of *Distributed Systems*, 2nd Ed.
- Leslie Lamport, *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM 21(7): 558-565 (1978)
- F. Mattern, "Virtual Time and Global States of Distributed Systems", in Proc. Workshop on Parallel and Distributed Algorithms, Elsevier, pp. 215-226.
- C. Baquero and N. Preguiça, "Why logical clocks are easy", Communications of the ACM 59(4): 43-47 (2016)

Fault Tolerance

State Replication with Primary Backup

Pedro F. Souto (`pfs@fe.up.pt`)

May 12, 2021

Replication: What and Why?

What? **Replication** is the use of the multiple instances/copies of processes/data, that we call **replicas**

Why?

Availability If one replica is down or unreachable, we can access the other replicas

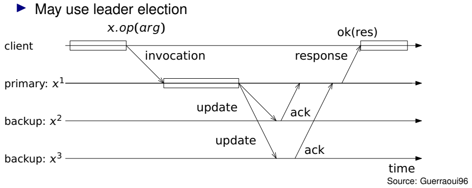
Scalability We can share the load among the replicas, and therefore handle higher loads by adding new replicas

Performance By accessing a replica that is closer, a client will experience a lower latency

It is not easy to achieve all of these simultaneously

Primary Backup Replication: Basic Algorithm

- One server is the **primary** and the remaining are **backups**
- The clients send requests to the primary only
- The primary executes the requests, updates the **state** of the backups and responds to the clients
  - After receiving **enough** acknowledgements from the backups
- If the primary fails, a **failover** occurs and one of the backups becomes the new primary.
  - May use leader election



Primary-Backup: Failure Detection and Failover

Failure Detection

How? Usually sending:

either, I'M ALIVE messages periodically

or, acknowledgment messages

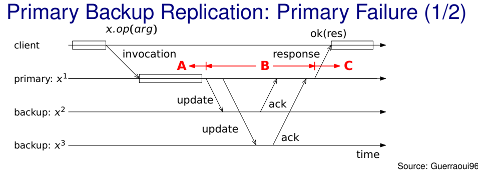
How reliable is it?

- It isn't, unless the system is synchronous ...

Failover

- At least, "select" new primary

Primary Backup Replication: Primary Failure (1/2)



What if the primary fails?

Depends when the failure occurs

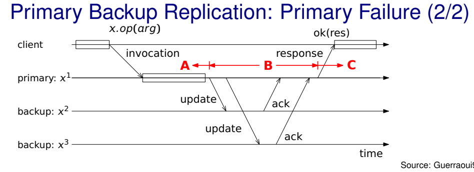
Primary crashes after sending response to client (C)

- Transparent to client
  - Unless response message is lost, and primary crashes before retransmitting it (case B)

Primary crashes before sending update to backups (A)

- No backup receives the update
- If client retransmits request, it will be handled as a new request by the new primary

Primary Backup Replication: Primary Failure (2/2)



Primary crashes after sending update (and before sending a response) (B). Need to consider different cases:

No backup receives update as in case A

All backups receive update

- If client retransmits request, new primary will respond
  - Update message must include response, if operation is non-idempotent

Some backups, not all, receive update

- Backups will be in inconsistent state

Must ensure update delivery atomicity

Primary Backup Replication: Recovery

Problem when a replica recovers, its state is stale

- It cannot apply the updates and send ACKS to the new primary

Solution Use a **state transfer** protocol to bring the state of the backup in synch with that of the primary

State transfer protocol Two main alternatives

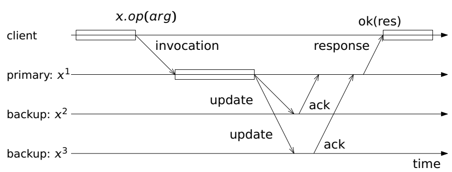
Resending missing UPDATES

Transferring the state itself

In both cases, the recovering replica can:

- Buffer the UPDATE messages received from the primary
- Process these UPDATES once its state is sufficiently up to date, i.e. reflects all previous UPDATES
  - Update the local replica
  - Send ACK to the primary

Primary-backup fault-tolerance



Question What's the fault-tolerance?

Answer It depends on the failure model

Crash-failure Two faulty replicas

- In general, $n - 1$

Omission In this case, there is a need for a majority to prevent the existence of more than one primary at some time instant

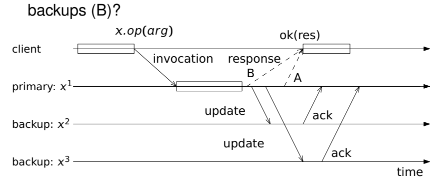
Primary Backup Replication: Non-blocking Algorithm

Observation Waiting for backup acknowledgements increases latency

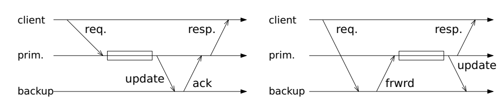
Solution Primary may send response to client before receiving `ack`'s from backups (A)

Question 1 What is the trade-off?

Question 2 What about sending response before the update to the backups (B)?



Primary Backup, with one Backup (Alsberg and Day)



Failure detection need to prevent "split-brain"

- E.g. use redundant links between replicas
- Or else use the same network interface to communicate with clients and other replica
  - This way faults caused by the network interface should affect the communication with both the other replica and the clients

Question Can we change the order in which the update and the response are sent in the RHS image?

Further Reading

- van Steen and Tanenbaum, *Distributed Systems*, 3rd Ed.
  - Section 7.5.2: Primary-Based Protocols
- R. Guerraoui and A. Schiper, *Software-based replication for fault-tolerance*, in IEEE Computer, (30)4:68-74 (April 1997)(in Moodle)

Fault Tolerance

Consensus

Pedro F. Souto (`pfs@fe.up.pt`)

May 18, 2021

Distributed Agreement: Informal

Problem How to ensure that the processes in a group agree on the actions to take?

Observation This is rather ambiguous. What do we mean by **agree**?

- Doesn't atomic commitment require agreement?
- There are actually a few problems very similar, but that are nevertheless different
  - We need to be more rigorous.

Consensus: Formal

Def. Consider a set of processes $1, \ldots, n$

1. Each process starts with an input from a fixed value set V
2. The goal is that each process choose a value in V
3. If a process chooses a value v, that decision is irreversible
4. The values chosen by each process must satisfy the following assertions

   Agreement Only a single value can be chosen by all processes

   Validity If all processes have the same input value v, no value different from v can be chosen

   Termination In any failure free execution, eventually all processes chose a value

The Synod Algorithm: A solution for Consensus (1/6)

Assumptions

Processes
- Operate asynchronously
- May fail and recover
- Have access to stable storage

Messages
- Delays are unbounded
- Messages may be lost or duplicated
- Messages are not corrupted

But Given the impossibility of consensus in asynchronous systems is well known (FLP)

"We won't try to specify precise liveness requirements. However, the goal is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value."

The Synod Algorithm: Roles and Structure (2/6)

Process Roles

Proposers Send **proposals** to the acceptors

Proposal is a pair (n, v), where n is a unique number (a proposal identifier) and v is some value from V;

Acceptors **Accept** the proposals
- A proposal is accepted if it is **valid** (to be defined)
- A value is **chosen** if a **majority** acceptors accept a proposal with a given value

Learners Learn the **chosen** values

Structure

Phase 1 Find a number that makes the proposal likely to be accepted
- And the value that has been chosen, if any

Phase 2 Submit a proposal

The Synod Algorithm: Phase 1 (3/6)

Phase 1

Proposer Selects a proposal number n and sends a PREPARE(n) request to a majority of acceptors

Acceptor Upon receiving a PREPARE. If:

n is larger than any previous PREPARE request to which it has already responded, then it responds with:
1. **a promise** not to accept any more proposals numbered less than n and
2. with the highest-numbered proposal (if any) that it has **accepted**

Note If n is not larger than that of a previous PREPARE request to which it has already received, then the acceptor does not need to respond
- How does then a proposer learn about a larger number?

The Synod Algorithm: Phase 2 (4/6)

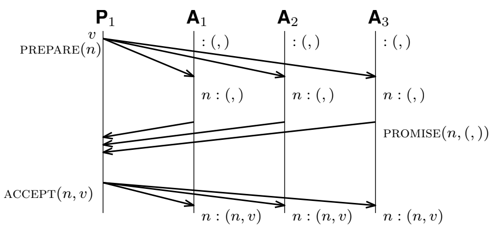
Phase 2

Proposer If it receives a response to its PREPARE requests (numbered n) from a **majority** of acceptors, then it sends an ACCEPT(n, v) request to each of those acceptors for a **proposal** numbered n with value v, where v is:
- the value of the highest-numbered proposal among the responses received in phase 1
- or is the proposer's input value, if the responses reported no proposals

Acceptor If an acceptor receives an ACCEPT request for a proposal numbered n, it accepts the proposal **unless** it has already responded to a PREPARE request having a number greater than n

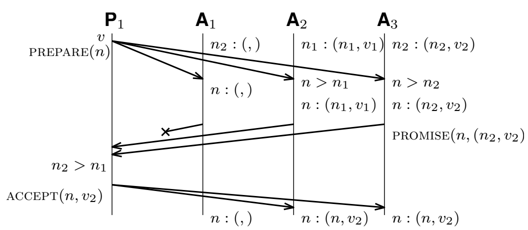
Note The rule to choose the value of a proposal is crucial to ensure that if a value has been chosen then higher-numbered proposals will propose that value

Synod's Execution: Simple



- The PREPARE message does not include any proposed value

Synod's Execution: More Interesting



- Should acceptor $\mathbf{A}_1$, also accept the proposed value?
  - If the proposer also sent $\mathbf{A}_1$ the ACCEPT request

The Synod Algorithm: Correctness Arguments

- If a value is **chosen**, it must have been accepted by a **majority** of the acceptors
- **Once a value has been chosen**, if a proposer gets a response to its PREPARE message from a majority, then the **highest-numbered proposal** it receives in the responses will have the **chosen value**
  - This can be proven by induction
- Therefore, **once a value has been chosen**, **every proposal** submitted (in phase 2), will have the value chosen
- Hence, **once a value has been chosen, every accepted proposal** will have the chosen value

The Synod Algorithm: Learning a Chosen Value (5/6)

- To learn the chosen value, the learner must find out that a **proposal** has been accepted by a majority of acceptors
  1. An acceptor may notify all learners every time it accepts a proposal
     - This minimizes the learning delay
     - But may generate too much traffic
  2. A learner may learn about the acceptance of the proposal from another learner
     - We are assuming non-Byzantine failures
     - Acceptors may notify a distinguished learner, which in turn notifies the other learners
     - The distinguished learner may be chosen by some election algorithm
     - Alternatively, a set of distinguished learners could be used
  3. Because of the loss of messages a learner may not learn about a chosen value.
     - The learner can ask the acceptors what proposals they have accepted, but the failure of an acceptor may make it impossible for a learner to find out if a proposal was accepted by a majority
     - An alternative, is to have a proposer to issue a proposal, using the algorithm described

The Synod Algorithm: Ensuring Progress (6/6)

- Two or more proposers may enter a kind of livelock situation in which the PREPARE from one prevents the other's proposal from being accepted
- To ensure progress, a distinguished proposer must be selected as the only one to try issuing proposals
  - If it finds out that its proposal number is not high enough, e.g. in case there is more than one leader, it can eventually choose a high enough proposal number
- FLP's impossibility result implies that leader election must use either randomness or real time, e.g. timeouts
  - However, the synod algorithm ensures safety regardless of success or failure of the election

Paxos: an Implementation of the Synod Algorithm

- Each process plays the role of proposer, acceptor and learner
- The algorithm chooses a leader, which plays the role of the:
  - The distinguished proposer
  - The distinguished learner
- The messages exchanged are those described
  - Responses are tagged with the corresponding proposal mumber
- Stable storage is used to keep state used by acceptors and that must survive crashes
  - The largest number of any PREPARE to which it responded
  - The highest-numbered proposal it has accepted
  - Updates to stable storage must occur before sending the response
- Uniqueness of proposal numbers is ensured by using a pair (cnt, id) where id is the the proposer's id and cnt is a local counter
  - Each proposer stores in stable storage the highest-numbered proposal it has tried to issue

State Machine Replication

What is? A generic approach to develop a fault-tolerant system, originally proposed by Lamport in his 1978 paper, "Time ..."

Idea
1. Design a service as a determinitistic state machine, which
   - Changes its state (variables)
   - Produces an output

   upon execution of an (atomic) operation
2. Replicate that state machine in different nodes
3. Set all replicas to the same initial state
4. Execute the same sequence of operations in all SM replicas

Challenge How do you ensure that all replicas execute the same sequence of operations?
- Clients may submit different operations "simultaneously"

Solutions
1. Use atomic (total order) multicast
2. Execute Paxos (consensus)

State Machine Replication with Paxos (1/5)

Idea Run Paxos to decide which command should be command n
- The i^{th} instance is used to decide the i^{th} command in the sequence
- Different executions may be run concurrently
- But must ensure that if the same command is chosen by more than one execution, it is executed only once

Normal Operation
- A single server is elected to be the leader, wich plays the role of distinguished proposer
- Clients send commands to the leader
- The leader:
  - Chooses a sequence number for the command
  - Runs an instance of Paxos for that sequence number, proposing the execution of that command
- The leader may not succeed, if:
  - It fails
  - Another server believes itself to be leader

State Machine Replication with Paxos (2/5)

- Initially, the servers elect a leader
- The **first** leader executes phase 1 for **all instances**
  - This is a particular case. We'll see the general case below.
- This can be done using a single very short message
  - Just use the **same proposal number**, e.g. 1, for all instances
- An acceptor will respond with a simple OK message
  - This is a particular case. We'll see the general case below
- After that, as the leader receives clients commands it can run phase 2 for the **next** instance
- A leader may start phase 2 of instance i before it learns the value chosen by instance $i-1$
  - An implementation may bound the number of **pending** phase 2 instances, i.e. phase 2 instances for which the leader has not learned the chosen value
- A server may execute command i iff:
  1. It learned the command chosen in the i^{th} instance
  2. Has executed **all** commands up to command i

State Machine Replication with Paxos (3/5)

Observation Because of the concurrent execution of multiple phase 2 instances, when a leader fails, the new leader may have not learned the value chosen for some instances:
- It may have learned the command chosen for instance i but not for instance $i-k$, where k is less than the size of the window of the pending phase 2 instances
- It may even be the case that no value has been chosen yet

State Machine Replication with Paxos (4/5)

General Case Initially, a new leader executes phase 1 for **all** instances whose chosen values it has not learned yet
- This can be done using a single PREPARE message, which:
  - specifies all instances for which the new leader has not learned the chosen value
  - specifies the same proposal number for all of them
- An acceptor responds with a single message which
  - includes any accepted proposal for the instances specified in the request
- If no value has been chosen to some of the pending phase 2 instances yet, the new leader may propose a NO-OP operation for those instances

Observation If failure of a leader is a rare event, the cost of executing a state machine command is essentially the cost of executing only phase 2.
- This is the best we can hope for (it has been proven): Paxos is **almost optimal**

State Machine Replication with Paxos (5/5)

What if leader election fails?

No leader is elected no new commands will be proposed

Multiple servers think they are leaders they can all propose values in the same instance of the consensus algorithm
- This may prevent any command from being chosen
- But safety is preserved: two servers will never disagree on the value chosen as the i^{th} state machine command

Conclusion
- Election of a single leader is needed only to ensure progress

Observations
1. Actually, this solution is essentially an implementation of atomic reliable broadcast using consensus (Paxos)
2. Given the assumption of process recovery, this description is somewhat incomplete:
   - The messages sent while a process is down must be delivered in order when the process recovers

Further Reading

- Leslie Lamport, **Paxos Made Simple** in *ACM SIGACT News (Distributed Computing Column)* 32, 4 (December 2001) 51-58.
  - Leslie Lamport, **The Part-Time Parliament** in *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133-169.
- T. Chandra, R. Griesemer and J. Redstone, **Paxos Made Live - An Engineerign Perspective**, in *ACM PODC'07*, 2007, 398-407
- R. van Renesse, **Paxos Made Moderately Complex**, 2011
- F. Schneider, **Implementing fault-tolerant services using the state machine approach: A tutorial**, in *ACM Computing Surveys 22, 4 (December 1990), 299–319*

Fault Tolerance
Group Based Communication

Pedro F. Souto (pfs@fe.up.pt)

May 27, 2021

Groups

Observation The concept of group of processes is recurrent in distributed systems
1. IP multicast groups
2. Garcia-Molina's Invitation Algorithm uses process groups for leader election in asynchronous systems
3. The state machine replication approach uses a group of processes to provide a fault-tolerant service by masking process failures

Types
- Static The group membership does not change (SMR with Paxos)
- Dynamic The group membership changes (Invitation Algorithm)
  - As processes join/leave the group voluntarily
  - As processes fail/recover

Observation Use of process groups is more convenient when integrated with group communication
- I.e. multicast

Dynamic Group Communication

Observation SMR with Paxos relies on total order multicast in a static group
- Implemented on top of the Paxos algorithm

Dynamic Group Communication relies on two services
- Group membership which provides information on which processes belong to the group
- Group communication which provides group-based messaging services
  - More precisely reliable multicast communication

Group Membership (1/2)

Basic Service Outputs a **view** of the group
- Each view is composed of a set of processes
- Each view has an identifier, v_i
  - Allows to distinguish among groups with the same composition
  - An alternative approach, is to ensure that a process gets a new identifier every time it joins the group
- If a process has a view:
  - All processes in that view must have agreed to join the view

Type
- Primary component Ensures that at "any time" there is at most one view
  - More precisely, the views are totally ordered
  - This is achieved by requiring a view to comprise a majority of the processes
- Partitionable Allows the existence of more than one view at any time

Group Membership (2/2)

Interface
- `join/leave` Used by processes to request joining/leaving groups
- `new-view` Used to notify a view change, either in response to:
  - Voluntary requests (join/leave)
  - Unexpected events (failure or recovery of processes)

Failure Detection needs not be reliable
- A process may be expelled from a group by mistake
  - E.g. because of transient communication problems
- If churn is too high, progress may be affected

Reliable Broadcast

Question What does it mean to **reliably broadcast** a message ?

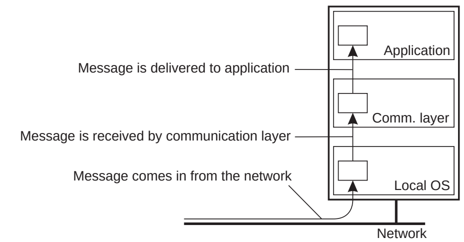
Assuming static groups first
- Validity If a correct process broadcasts message *m*, then all correct processes in the group deliver *m* **eventually**
- Agreement If a correct process delivers message *m*, then all correct processes in the group deliver *m* **eventually**
- Integrity A process delivers message *m* at most once, and only if it was previously broadcasted by another process
  - In the case of **closed groups**, the broadcaster must be a group member too.

Failure assumptions
- Processes may fail by crash and may recover
- Any network partition **eventually** heals

What if the group is dynamic?

Delivering vs. Receiving

- **Delivering** a message is different from **receiving** it:



Reliable Broadcast in Dynamic Groups (1/2)

- Simplify first ... assume **closed group communication**
- Each multicast message is associated with a group view
- **First attempt**: reliable broadcast properties
  - Validity If a correct process broadcasts message *m*, then all correct processes in the group deliver *m* **eventually**
  - Agreement If a correct process delivers message *m*, then all correct processes in the group deliver *m* **eventually**
  - Integrity A process delivers message *m* at most once, and only if it was previously broadcasted by another process

  must hold only for members of that group view:
  - Validity If a correct process broadcasts message *m* in a view, then all correct processes in **that view** deliver *m*
  - Agreement If a correct process **in view** *v* delivers message *m*, then all correct processes in that view deliver *m* **in view** *v*
  - Integrity No need to change
- But ... we are not there yet.

Reliable Broadcast in Dynamic Groups (2/2)

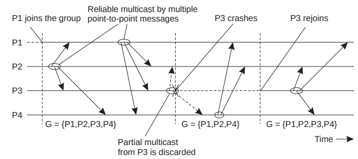
Challenge **Validity** and **Agreement** require all correct processes to deliver a message. This **conflicts** with:
- Groups as a mean to keep track of the state of processes in the system in a coordinated way;
- Impossibility of distinguishing between:
  - Slow processes;
  - Failed processes;
  - Unreachable processes.

  i.e. accurate and complete process failure detection in an asynchronous system

Scenario Assume a clean network partition
- Consider a sender in one of the parts
- Consider the processes in another part

View/Virtual Synchronous Multicast

Virtual Synchrony If processes *p* and *q* change from view *V* to view *V'*, then they deliver the same set of messages in view *V*
- This is a variation of **agreement**
- It can also be seen as an **atomicity** property
  - Either a message is delivered to all processes or ...



Self Delivery If a correct process *p* broadcasts message *m* in view *v*, then it delivers *m* in that view
- This is a variation of **validity** and precludes trivial solutions such as a protocol that never delivers messages, even ...

View Synchronous Multicast: Implementation (1/4)

Assumptions/Model

Point-to-point channels Communication is via point-to-point channels

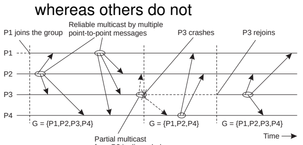
Reliable channels If the processes at the ends of a point-to-point channel are correct, then a message sent in one end will be delivered at the other
- It is well known how to achieve this by acknowledgments/retransmissions
  - As long as communication failures are not "too frequent"

FIFO channels I.e. messages are delivered in order
- There are well known techniques

Crash-failures Processes fail by crash only

View Synchronous Multicast: Implementation (2/4)

Problem What if the sender fails in the middle of a multicast
- In this event, some processes may **receive** the message whereas others do not



Solution Two alternatives:
1. Deliver the message only if all correct processes receive it
   - Increases the delivery latency
2. Deliver the message immediately
   - Upon a view change, processes that survive the current view must send each other the messages they have delivered that may have not been received by other view members

View Synchronous Multicast: Implementation (3/4)

Definition A **message** *m* **is stable** for process *p*, if *p* knows that all other processes in the view have received it

Idea
- Keep a copy of the messages delivered until they become stable
- Upon a view change:
  1. Resend all non-stable messages to the remaining processes
  2. Wait for the reception of non-stable messages from other processes
     - and deliver them if they have not been delivered yet
  3. Change to the new view

  Alternatively a process may be elected as **coordinator**
  1. each process sends its non-stable messages to the coordinator
  2. the coordinator then broadcasts each of them

  Observation Election does not require extra messages
  - given that the group members are known, we can use some *a priori* rule – e.g. the process with the smallest identifier

View Synchronous Multicast: Implementation (4/4)

Problem How does a process know that all non-stable messages have been received?

Solution
- Each process, sends a FLUSH message, after sending all non-stable messages
- Upon receiving a FLUSH message from each process that is in both the current and the next view, a process may change its view

Problem What if the processes crash during this protocol

Solution Processes must start a new view change

Problem How do you ensure progress?

Order in Multicast Communication

Observation Like in unicast communication, order is orthogonal to reliability
- Must be careful in the definitions so that we keep them that way

Unordered no guarantee on the order in which messages are delivered

FIFO if messages m_1 and m_2 are sent by the same process in that order, then if a receiver delivers both of them, it must deliver them in that order

Causal if message m_2 "causally depends" on message m_1, then if a receiver delivers both messages, it must deliver m_1 first

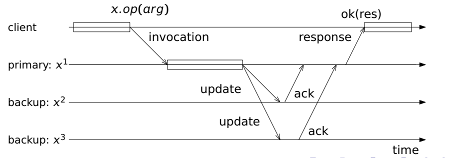
Total if process *p* delivers message m_2 after message m_1, then if process *q* also delivers both messages, it must deliver them in that order

Further Reading

- Tanenbaum and van Steen, *Distributed Systems*, 2nd Ed.
  - Section 8.4: Reliable Group Communication
  - Section 7.5.2: Primary-Based Protocols
- K. Birman, A. Schiper and P. Stephenson, *Lightweight Causal and Atomic Multicast*, ACM Transactions on Computer Systems, Vol. 9, No. 2, Aug. 1991, Pages 272-314

Primary Backup Replication: Basic Algorithm

- One server is the **primary** and the remaining are **backups**
- The clients send requests to the primary only
- The primary executes the requests, updates the **state** of the backups and responds to the clients
  - After receiving enough acknowledgements from the backups
- Essentially, the primary orders the different client requests
- If the primary fails, a **failover** occurs and one of the backups becomes the new primary.
  - May use leader election



Source: Guerraoui96

Primary-Backup: Failure Detection and Failover

Failure Detection

How? Usually sending:

either, I'M ALIVE messages periodically

or, acknowledgment messages

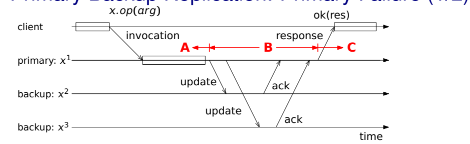
How reliable is it?

- It isn't, unless the system is synchronous ...

Failover

- At least, "select" new primary

Primary Backup Replication: Primary Failure (1/2)



Source: Guerraoui96

What if the primary fails?

Depends when the failure occurs

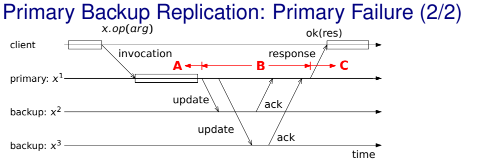
Primary crashes after sending response to client (C)

- Transparent to client
  - Unless response message is lost, and primary crashes before retransmitting it (case B)

Primary crashes before sending update to backups (A)

- No backup receives the update
- If client retransmits request, it will be handled as a new request by the new primary

Primary Backup Replication: Primary Failure (2/2)



Source: Guerraoui96

Primary crashes after sending update (and before sending response) (B). Need to consider different cases:

No backup receives update as in case A

All backups receive update

- If client retransmits request, new primary will respond
  - Update message must include response, if operation is non-idempotent

Some backups, not all, receive update

- Backups will be in inconsistent state

Must ensure update delivery atomicity

Primary Backup Replication: Recovery

Problem when a replica recovers, its state is stale

- It cannot apply the updates and send ACKS to the new primary

Solution Use a **state transfer** protocol to bring the state of the backup in synch with that of the primary

State transfer protocol Two main alternatives

Resending missing UPDATES

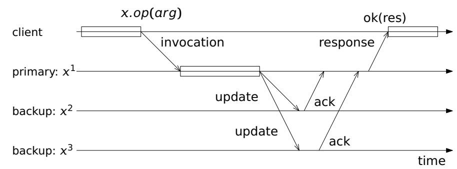
Transferring the state itself

In both cases, the recovering replica can:

- Buffer the UPDATE messages received from the primary
- Process these UPDATES once its state is sufficiently up to date, i.e. reflects all previous UPDATES
  - Update the local replica
  - Send ACK to the primary

Similar issues arise with SMR

Primary-backup fault-tolerance



Question What's the fault-tolerance?

Answer It depends on the failure model

Crash-failure Two faulty replicas

- In general, $n - 1$

Omission In this case, there is a need for a majority to prevent the existence of more than one primary at some time

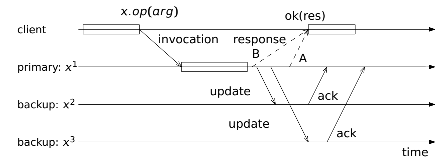
Primary Backup Replication: Non-blocking Algorithm

Observation Waiting for backup acknowledgements increases latency

Solution Primary may send response to client before receiving `ack`'s from backups (A)

Question 1 What is the trade-off?

Question 2 What about sending response before the update to the backups (B)?



PBR Implementation with View Synch. Mcast (1/4)

Idea The primary can be determined from the view membership without further communication

- The primary sends the updates to the backups using view synchronous multicast (ensures message delivery atomicity)
  - What about order?

Primary

1. Upon receiving a request, the primary:
   1.1 executes it
   1.2 generates an **update** representing the change in the state
   1.3 multicasts the UPDATE to the current view, v_i
      - must include **request id** and **client response**, unless ...
2. Upon receiving an ACK from f backups
   - sends the reply back to the client

Backup

1. Upon receiving an UPDATE, the backup:
   1.1 updates its state accordingly
   1.2 sends its acknowledgement to the sender (via multicast?)

PBR Implementation with View Synch. Mcast (2/4)

How does VSynch Mcast help?

Answer

- Upon failure of the primary generates a new view, and "elects" a new primary
  - A new view is also generated upon
    - either a failure
    - or the recovery

    of a backup
- Ensures UPDATE delivery atomicity to replicas that move from one view to the next
  - For every UPDATE, either all replicas that move from that view to the next deliver the UPDATE or none does it.

PBR Implementation with View Synch. Mcast (3/4)

What VSMcast does not address?

Upon a view change new members must synchronize their state

- Still need a state transfer protocol

At most-once semantics i.e. process a request no more than once

- TCP is no solution. Why?
- If uncertain:

1. Cache the response
   - Need to do it to recover from lost messages anyway
   - But backups also need to know the response
2. If the client retransmits the request, resend response

PBR Implementation with View Synch. Mcast (4/4)

The devil is in the details VSC simplifies significantly

- But replica reintegration is not trivial, even with VSC
- Paxos also glosses over the issue of recovery

For a detailed discussion of recovery somewhat application dependent

- B. Liskov, *From Viewstamped Replication to Byzantine Fault Tolerance*, Ch. 7 of LNCS 5959
  - **Viewstamped replication** is an algorithm that
    - Uses the concept of view, like VSC
    - But is more asynchronous, like Paxos

Further Reading

- van Steen and Tanenbaum, *Distributed Systems*, 3rd Ed.
  - Section 7.5.2: Primary-Based Protocols
- R. Guerraoui and A. Schiper, *Software-based replication for fault-tolerance*, in IEEE Computer, (30)4:68-74 (April 1997)(in Moodle)
- R. Guerraoui and A. Schiper, *Fault-Tolerance by Replication in Distributed Systems - A Tutorial* International Conference on Reliable Software Technologies (Invited Paper), Springer Verlag (LNCS1088), 1996