2017-2 Distributed Systems Part1

30866 characters in 4713 words on 838 lines

Florian Moser

February 20, 2018

1 motivation and history

1.1 distributed systems

multiple autonomous processors that do not share primary memory
cooperate by sending messages over a communication network

abstractions of distributed systems
network with nodes (routing, addressing)
objects provided by OS, middleware, languages (client/server API)
algorithm and protocols (actions, events, consistency, correctness)

distributed systems connect
systems (to use resources jointly)
functions (for cooperation in usage of specialized resources)
resources (to combine capacities)
data (to make it globally accessible)
redundant systems (survival)

concepts
concurrency, synchronization
programming languages as communication objects
parallel / distributed algorithms
semantic of cooperation and communication
abstraction principles
basic phenomena of distribution

examples
physically distributed computer cluster, network
logically distributed processes (each own state, time)
electronic commerce, communication

historical
computer-computer communication (data transfer, master-slave)
ARPNET (peer to peer)
workstations (LAN)
commercial pioneer projects (banks, flight reservation systems, WAN)
web/internet (eCommerce, web services)
mobile devices (smartphone, WLAN)
internet of things (door, refrigerator)

1.2 problems

general problems
heterogeneous software and hardware
partial failures possible (instead of total failure)
security difficult to enforce but more important than single-user

snapshot problem
need global view despite continuous ongoing changes

phantom-deadlocks
in t = 1, B waits C; observing B determines that B waits C
in t = 2, A waits B; observing A determines A waits B
in t = 3, C waits A; observing C determines C waits A
looks like a deadlock but observations done at different times
need to detect such problems

clock synchronization
how to evaluate clock offset / different running speed?
need to synchronize clocks at different devices

inconsistencies
causal observations difficult
hole makes pressure decrease therefore pump increases power
observer might see increase before pressure drop
observer therefore might assumes the pump made a mistake

secret establishment over insecure channels
idea that it may works gives lock example
a sends secret with own lock to b
b adds its lock and sends it back
a removes lock and sends it to b
b can now remove its own lock
need way to make this possible in software

2 ARCHITECTURES

2.1 architectures of distributed systems

monolithic
terminals give commands (mainframes)

peer-to-peer
node is provider and consumer (ARPNET)

client-server
server as provider, client as consumer (internet)

fat- or thin client
depending on where you do presentation/application/data logic
some presentation must be at client, some data must be at server

3-tier
processing is distributed to multiple entities divided logically
easier maintenance, easier replacements, optimized hardware

multi-tier
more layers help with scaling and flexibility
better computation distribution
distributed databases help with replication
only possible because hardware is so cheap

compute cluster
concentrated into small space (few meters) with fast interconnectivity
different net topologies for different use cases

cloud computing
concentrate computational power at a central place, outsource applications
no maintenance, everywhere available, no data backups
cheap because of scaling effects
can adapt to changes in business requirements
in the future, cloud unit container parked close to power plants

parallel vs distributed system
coupling is the distinctive factor
parallel systems are multicores (same chip) with shared memory
distributed systems are compute cluster and compute networks

2.2 net topologies

hypercube
die of dimension d
easy routing (XOR with receiver, simply flip bit at each node), short paths
but needs a lot of connections (n log n)

d-dimensional torus
construct by taking w elements of dimension d-1 and connect
corresponding elements to ring
wrap-around grid

3 communication

3.1 cooperation by exchanging messages

to cooperate processes they need to exchange information
use shared memory or send messages
messages need processing power and management

required
physical medium in between
clear defined behaviours
common language and semantic

implicit communication
receiver can infer from actions of sender how far it progressed

message passing system

organizes transport, and manages resources
implements higher communication protocols
guarantees certain properties (priorities, in-order receive)
masks mistakes (timeouts, AKS, sequencing, repeat, ...)
hides heterogeneity of different systems (eases portability)

3.2 properties

in-order receive (FIFO)
send order = receive order
but allows for messages to be indirectly surpassed
A sends to B, A send to C, B sends to C
C receives from B, C receives from A

in-order receive (causal ordering)
send order = receive order
but no message is allowed to indirectly surpass another
generalizes FIFO to all processes

priority
unclear semantics
how to process high priority messages?
how to ensure fairness and neutrality?
why not just ignore priority of messages?
possible applications are pause/abort of actions, break of deadlocks, ...

failure modes
send/receive/transfer failure (bad connection)
crash/fail-stop of process (404 errors)
time failure (event happens to too late or too early)
byzantine / rogue processes with invalid messages / behaviours
some can only be observed using redundancy

3.3 communication types

message oriented
unidirectional
fire & forget
sending process can continue working directly after sending message

task oriented
bidirectional
result of request will be passed back to sender
client waits till response received, till task finished at receiver

3.4 processing types

blocking send
sender waits till transaction is finished
sender has guarantee that request has been received / processed
receiver sends ACK when request is received (message)
receiver sends ACK when request is processed (task)

non-blocking
informs communication system of available requests
but does not wait for sending
returns handler which can be queried if message has been sent

3.5 buffer

sits between sender & receiver, has own process

if new message received from sender
can wait for another message
can wait in blocking send for receiver

implementation with proactive receiver
receiver asks buffer for new message whenever ready
receives no response if buffer empty
if buffer full it stops accepting messages from sender

implementation as multi-thread object
with buffer ring, FIFO
buffer is in shared address space of sender and receiver

3.6 synchronous communication

idealized view is that send & receive happen at the same time
send operation returns after message was delivered to receiver
can be implemented with blocking send
can simulate async using buffer

3.6.1 receiver / sender waits

receiver blocked till message is inbound
sender frozen till receiver ready, processed message, ACKed

3.6.2 virtual simultaneity

create diagram with lines containing senders as dot
add messages as arrows from sender dot to receiver dot
move around dots without changing order till all arrows are vertical
virtual simultaneity fulfilled if no arrows cross at end of transformation

3.6.3 deadlocks

if cyclic dependency in wait-for-graph
A waits for B, B waits for A

3.6.4 advantages compared to async

sender does knows when/if message has been received
debugging is easier

3.6.5 implementations

RPC (task)
executes task on other machine, waits for result
handy for programming as behaves as any other method call
RPC (remove procedure call)

rendezvous (message)
sender repeatedly contacts receiver till no more NACK received
or sender sends message which is put in buffer at receiver
or receiver sends ACK to sender as soon as he is ready
needs small buffers only, but busy waiting & complex protocol

3.7 async communication

no guarantee that message has been delivered/processed
can simulate sync by waiting for explicit acknowledgement

3.7.1 advantages compared to sync

sending process can continue while message is send over networks
less coupling between sender and receiver (can be unresponsive)
higher degree of parallelism
less danger of communication deadlocks

3.7.2 implementations

asynchronous RPC (task)
also called Remove Service Invocation
parallelization of sever/client possible
to implement use await, callbacks, future-variables

no-wait send (message)
sender is only blocked till message is on its way
very fast if not buffer full or other sending issues
server/client are properly separated & simple implementation
but unknown if message has been received, buffer overhead

3.8 RPC

like a procedure call
clear semantics for executor
simple to program in high-level API's (like any other method call)
abstract complexity due to distributed factors as good as possible

3.8.1 example call

client calls procedure, stubs marshal, transport sends request
server receives request, stubs unpack arguments, local procedure call
server produces result, stubs marshal, transport sends reply
client receives reply, stubs unpack result, result is returned

3.8.2 stubs

take care of packing/unpacking (converting representations)
set timeouts, raise exceptions, pass messages
simulate "local" procedure call
can be generated

3.8.3 capability of data structures

how to convert representations?
numbers (big endian / little endian)
characters (UTF8 / ASCII)
types like strings (length / '/0')
arrays (row / column wise)

3.8.4 marshalling

creating of message from parameters
flattening complex objects
use representations the other party understands

3.8.5 conversion

converting of objects in common notations, for example as XML or "receiver makes it right" (send whatever, receiver has to correct)

3.8.6 transparency

RPC should behave as local procedure calls
not always possible (server/network failure, difference in live cycles)

3.8.7 performance transparency

RPC's slower than real local procedure call
communication size can be quite big
sudden delays possible

3.8.8 performance analysis

transport cheap
conversion (as headers, checksums) is expensive
copying is expensive
context-switch is relevant when using small messages

3.8.9 place transparency

target must be named explicitly
no global variables
no pointers/references

3.8.10 callback RPC

temporary role reversal
client receives status updates from server

3.8.11 context handles

structure which contains context information
enables server to remember client
is passed to client in reply, is included in the next request

3.8.12 broadcast/multicast

request is sent to other servers at the same time
broadcast sends to all, multicast only to some
RPC is finished after first response (or client can wait for more results)

3.8.13 security

authentication when creating connection ("binding")
authentication of each single request
end-to-end encryption of messages
make it impossible to modify (digital signature, checksums, MAC)

3.8.14 "secure RPC" as example

session key k encrypts messages
request contains encrypted timestamp
first request contains time window
server accepts request if timestamp bigger than last, if inside time window
server reply contains the last timestamp for client-side authentication
encrypted timestamp ensures attacker can't generate message without key
small time window ensures attacker can't bruteforce the key

3.8.15 failures

message can be lost (or too slow; can't be differentiated)
multiple failure causes, but mostly all-or-nothing behaviour
partial system fault (client or server) typical
different view of transaction state between server & client

3.8.16 failure transparency

missing request message
resend request after timeout
but how to choose timeout, how many retries, maybe server just too slow
multiple executions of command in resend request possible

missing reply message
same treatment as missing request, client can't know difference
server can cache replies, resend if same request received again
but how to clean up cache (time & reply ACK's)

server crash
client can't differentiate crash before, after, in procedure
maybe inconsistent server state after reboot

client crash / not longer interested
server waits indefinitely for ACK of client
blocks resources due to orphans at server
use "is-alive" ping while running procedures, discard old processes
or client explicitly contacts server for cleanup

3.8.17 failure semantics

maybe-semantic
no repetition of request, server may or may not answer
easy and efficient
useful for lookup services

at-least-once semantics
automatically repeat requests
stateless protocol on server side (no duplicates can be discovered)
nice for idempotent operations (reading a file)
maybe uses more resources than explicitly necessary

at-most-once semantic
automatically repeat requests
server identifies duplicates and may sends cached responses
nice for non-idempotent stuff
more expensive than at-least-once

exactly-once
not really possible
because if crashes occur no computations take place

3.9 communication concepts

ports
communication end point which abstracts structure of receiver
one process can have multiple ports

channels
name them, then send or read from them
broadcast with subscribers
very flexible because can change the connection structure any time

software bus
anonymous
can react to events
can send events

event channels
anonymous
can register for events
dispatches events
participants need to be always listening (maybe use buffers)

zeitüberwachter nachrichtenempfang
receiver sets max time he wants to wait, else other code is executed
also useful for blocking send

4 client-server

4.1 general

server provides infos
client consumes infos and provides front end for user

4.2 server

iterative server
will process one request at a time
take new request from buffer if finished with old
easy to realize, good for trivial stuff

concurrent server
concurrent processing of multiple requests

concurrent server with dynamic handlers
master creates slave "handler" for each request
may has fixed number of slaves ready for usage "process preallocation"
slave communicates directly with receiver
ceiling amount of slaves at the same time

stateless servers
every request must be fully described
HTTP theoretically stateless

state servers
can identify repeated requests, therefore idempotent
HTTP server may needs to identify customers

4.3 architectures

service oriented architecture (SOA)
service is more processing oriented (like photo service)
processing by calling external services with parameters
combines the results for the user

resource oriented architectures (ROA)
resource is more data oriented (like photos)
processing internally, but may uses external data

4.4 tasks

non-pure
like writing a file

pure ("zustandsinvariant")
simple lookups

idempotent tasks
repeated tasks lead to same result (but can be non-pure)

4.5 web stuff

identify customers
URL rewriting, dynamic webpages
cookie can be the context-handle
identify with IP (but not uniquely)

HTTP
possible protocol to transfer SOAP requests
GET (fetch), PUT (send), PATCH (change)
POST (replace), DELETE (delete)

4.6 lookup service

connects client & server
server makes itself known in LUS (lookup service)
client asks LUS and import the provided service configuration

pro
register multiple provides for same task for scalability
validate authorization
can use polling to see if server is still responsive
can manage multiple versions

contra
lookup needs time
LUS is single point of failure
clients need to know LUS

4.7 middleware

communication between application, hidding infrastructure complexity

4.7.1 RPC libraries

client-sever paradigm
easy interface, code generation
security such as authorization, authentication, encryption

4.7.2 client-sever distribution platforms

lookup service, global namespace, global filesystem
supported multi threading

4.7.3 object-based distribution platforms

cooperation between distributed objects
object-oriented interface
object request broker (ORB) functions as middleware

4.7.4 CORBA

ORB to redirect method calls
IDL interface description language with stub generation
CORBA update failed in 2000, different interests and better competition

possible methods calls
synchronous (waits for response)
delayed synchronous (can get object later)
one way (fire & forget)

4.8 SOAP

example for client-server model
internet is very homogeneous
web services define platform independent interface

4.8.1 keywords

HTTP (Hyper Text Transport Protocol) as transport layer
UDDI (Universal Description, Discovery and Integration) as lookup service
SOAP (Simple Object Access Protocol) specifies protocol
WSDL (Web Services Description Language) as service description

4.8.2 UUDI

currently not available cause money

4.8.3 protocol

envelope
body containing the data serialized as XML
header which may specifies additional options
transfered as HTTP body

engine
server stubs are generated from a webservice implementation (buttom up)
client stubs from WSDL description (top down)

example request specification
setDisplayPower([xsd:boolean])
[xsd:boolean] getDisplayPower()
can also use xsd:integer, xsd:string, xsd:complexType
can define own types [myCustomType], consisting of xsd properties

4.8.4 WSDL xml nodes

definitions
targetNamespace contains current element
xmlns:NS to add more namespaces

types
import other schemas, add own elements, add complexTypes

messages
can name messages, specifying the needed parameters

portType
describes a method
has operation sub nodes which describe input, messages and faults

binding
what protocol to use (HTTP, SMTP, UDP)
multiple bindings possible

service
where to access services
maps a binding to a concrete address (URL) for HTTP transfer

4.9 REST

uses URI (Unique Resource Identifier)
created for the web, as best way to use it

4.9.1 REpresentational State Transfer

not resource, but representations are transmitted
get access to state of resource, can alter & send them back

4.9.2 usage model

hypermedia as engine of application state
client knows only base URI
server broadcast other URIs per form or hyperlinks

4.9.3 example request

path /display/power
methods GET, PUT
representation text/plain
possible values true/false

4.9.4 principles

client-server
consists of components who can connect to clients, to server or both
User Agent which creates requests
Intermediary which redirects request potentially modifying them
Origin Server which has control of resources

statelessness
request contains all info for processing; context held client-side
crash/orphans less critical, easier scaling and monitoring, caching

caching
meta-data determines how long response is valid
clients/servers consult cache for answers with no further processing

uniform interface
addressing done with URI
requests are standardized (GET, POST, ...)
standard representations (XML, JSON, ...)
resourced provided in multiple formats, client chooses applicable

hypermedia as engine of application state
provide access point which allows discovery of URIs

layered system
clients don't know if connected to server or intermediate
intermediaries can be added at any point

code on demand(optional)
server can externalize logic to the client

4.9.5 properties

scalability
statelessness allows efficient servers / load balancing
caching reduces communications

adaptability
uniform interfaces decouple server & client
layering allows manipulation later
code on demand allows to update active clients

observability
requests which contain all infos are easily traceable

reliability
uniform interfaces & layering allows for redundancy

4.9.6 state

resource state
static templates & resources from server

client state
active rendered state & its history
bookmarks preserve full URI
back button of browser allows to go back to the prior state

statelessness
client & server state are strictly decoupled (hence sessions)
efficient, roboust against client/server crash
use url rewriting; encode client-specific information in requests

with state
can persist state over multiple sessions
potentially reduced transfer size
use cookies; server has client state, possibly changing request execution
problem back button; server/client state disjoint, URIs may stop working

5 Broadcast / Multicast

5.1 group communication

5.1.1 idealized memory based communication

all receive immediately
all receive at same time

5.1.2 pull

client requests infos from server
demand driven

5.1.3 push

server sends infos to client
event driven
client subscribes to channel, server publishes news

5.1.4 broadcast

send message to all members

5.1.4.1 challenges

network often not multicast, simulate by sending lots of messages
non-deterministic time shift, no sending guarantees
multicast protocol needs to approximate

5.1.4.2 lost messages

due to network overload, receiver not listening
receivers are not in the same state any more
need redundancy and complicated protocols to solve this

5.1.4.3 types

best effort broadcast
typically simple send without ACK
used to distribute non-critical information
used to implement higher protocols
very efficient if successful
no guarantees if and how many messages are delivered

reliable broadcast (with ACK)
waits for ACK for every single message
resends if none received
bad scaling because of polluting ACKs, need to distinguish duplicates

reliable broadcast (with NACK)
broadcasts contain identifier/sequence set by sender
receiver broadcasts missing messages with NACK, sender resends
sender can send empty messages to ensure receiver missed no messages
does not help if server / network crashes

reliable broadcast (flooding)
send message to all nodes except the originator
remember the sequence number of the message to avoid flooding twice
need only one connection to a not crashed server to receive the message

5.1.4.4 message ordering

5.1.4.4.1 FIFO

all broadcast messages from same sender are received in same order
does not imply causality

5.1.4.4.2 causal order

causality exists if there is a connection in space-time diagram from A to B
implies all messages are received according to the rules of causality

5.1.4.4.3 atomic

if two processes receive the same two messages, they are in the same order
does not imply FIFO & causal order

order atomic with central sequencing
unicast from sender to sequencer
broadcast from sequencer to other members
sequencer waits for ACK before sending next message

order atomic with token
single token created which contains sequence number
member with token can send message
token is passed around in predefined order
messages delivered according to sequencing number
new token generated if owner timeouts
use explicit token request instead of passing if a lot of members

5.1.4.4.4 causal + atomic

comparable with memory based communication
also called virtual synchronous communication
events happen at same logical time (which may not equals real time)
logical time only takes causality of messages into account
same as synchronous inside the system

5.1.5 multicast

send message to subgroup of members

why
simplify addressing
hiding of group assignment
logical unicast, groups have replaced individuals

hidden channels
messages which leave groups and return through another node
causually dependency for such messages must be defined

dynamic groups
members can join/leave group at any time
entry/exit should be atomic
senders should see active members at the time of send

5.1.6 tuple rooms

decouple sender and receiver
virtual, global storage
data can be added, changed, removed from all members

linda
language for tuple rooms
out(t) (adds), in(t) (reads & removes), read(t) (reads)
tuple room implemented as associative storage
get tuple by condition; ("hi", ?p) is tuple with "hi" as first attribute
asynchronous operations (readp and inp(t) do not block, return bool)
synchronous operations (read and in(t) wait for correct tuple to appear)

able to model server-client
client places requests and waits for responses
server processes requests and places responses
client; out("req", guid, params); in("resp", guid, ?result);
server; in("req", ?guid, ?params); out("resp", guid, result);

some tuple rooms support additionally
persistence (tuple will not perish after termination)
transaction (important if multiple servers access tuple room)

problems
central tuple room is weakest link
replicated / disjunct distributed tuple rooms
difficult for structured programming and verification

JavaSpaces
tuple room for java
can persist objects and behaviour
part of Jini (middleware for java)
can transport code to receiver, use common objects
ordering of operations between different processes undefined

5.2 logical time

time is useful to show causality, persist state, mutex

applications
state of system at specific point in time
show causality between events (if x before y, y did not cause x)
fair mutual exclusion (longest waiting is served)

real time
asymmetric, transitivity, irreflexivity, linearity, infinite, continuous (always point in between), metric, every point is eventually reached

causal relation (x<y) exactly when
x,y from same process and x before y
x is a send, and y its corresponding receive
there is a z for x<z and z<y
solve this with timestamps, called C(x)
if $e < e'$ then $C(e) < C(e')$ (time must imply causality)

logical clocks by lamport
fulfils weak clock consistency (if $e < e'$ then $C(e) < C(e')$)
at send, increase clock and send request
at receive, take max(own clock, received clock), increase clock
for total order use process id when same timestamp received

vector clocks
each process has its own counter (sizeof(vector) = count(processes))
fulfils weak and strong clock consistency ($e < e' \Leftrightarrow C(e) < C(e')$)
$C(e) < C(e')$ if all counter are smaller/equal, at least one smaller
at send, increase own clock and send request
at receive, take max of all clocks, increase own

6 MUTEX

6.1 mutex

conflict with unique resource

6.1.1 solution requirements

safety (nothing bad will ever happen, exclusive access guaranteed)
liveness (eventually something good will happen, progress)
fairness (all have to make progress, all profit)

6.1.2 manager

manager coordinates access, has queue of processes which are waiting
process sends "request", waits "grant", notifies afterwards "release"
simple, few messages
manager is single point of failure

6.1.3 global queue

replicate queue at each process
use FIFO queues, messages contain timestamp (real or Lamport)
requests and releases are sent to all, requests are confirmed with ACK

Lamport
3(n-1) messages
each member has own queue
use mutex if first in queue & received message from all others
request mutex with ("request", time), add to own queue
release mutex with ("release"), remove from queue
on receive of request, save time in queue, confirm with ("ACK")
on receive of release, remove entry from own queue

Ricart / Agrawala
2(n-1) messages
use mutex when received ACK from all other members
request mutex by broadcasting "request" with timestamp
on "request", send ack if (!self || sender_time < self_time)
else wait till released mutex

7 Security

7.1 requirements

authorization (only specific entities have access)
privacy (attackers can't read message)
authentication (sender is verified)
integrity (message is unmodified)
availability (no DoS possible)
non-repudiation (cannot deny the sending/reception of message)
prosecution (needs logging, need access to otherwise private keys)
compliance (conform to law, terms)
authenticity (of service, message, data)

7.2 security challanges in distributed systems

harder to guarantee security in distributed systems
no central security authority
systems often open which allows to easier spot possible attack points
standardized protocols are attackable as one can craft own packets
spatial distance makes it hard to locate attacker
heavy usage makes an attack more valuable
physical separation often not possible
tools such as wireless make it easier to launch an attack
heterogeneity allows more attack points
hard to enforce common security policy

7.3 authentication mechanisms

peer-authentication, ask question only associate can answer
password, but not tied to identity (sniffing, secrecy not enforcable)
one-way functions, but no mathematical proof such functions exist

7.4 attacks

passive attacks
observe communication
"who when with whom"
read messages

active attacks
modify messages (modify, remove, create, resend, delay)
impersonate (behave as another process, use foreign passwords)
malicious usage of services
deny usage of services with DoS

7.5 cryptography systems

encrypt with K1, decrypt with K2
asymmetric if K1 != K2
decryption is infeasible without the key
procedure should be public because difficult to keep secret, feedback useful

7.5.1 use biased random number generators

1 / 0 may have different probabilities
therefore only choose pairs of 01 (=0) or 10 (=1)
transform 01001101011110 → 01010110 → 0001

7.5.2 naming

n are nonces, random values used only once
m are messages
K are symmetric keys

7.5.3 symmetric keys (like DES, AES)

very fast, but key must be secret

one-time pad
perfect encryption
m_1 = m XOR pad, m = m_1 XOR pad (pad applied twice cancels it out)
pad must never be used twice, or repeated, must be real random numbers
not practical because need large amount of authenticated encryption bits

7.5.4 asymmetric (like RSA)

exchange keys easy (p public, s not exchanged, 2n keys for n members)
authenticates owner (if able to decrypt {m}_p authentication successful)
digital signature (if able to generate {m}_s authentication successful)
but slow

public key server
communication must be secure, no tampering, impersonation

public key service
distributes certificated public key and its private key to member
transfers session keys securely and authenticated to the members

properties
every member has (p,s) public key p, secret key s
m can't be derived from {m}_p
s can't be derived from p or {m}_p with known m, p
m = {{m}_p}_s

maybe additionally m = {{m}_s}_p

7.5.5 authentication

symmetric way
A and B share key K
A → B n
B → A m_1 = {n}_K
A verifies that {m_1}_K = n

asymmetric way (one way)
A → B n
B → A m_1 = {n, K}_sB
A decrypts m_1 with public key of B and now has K
safe against replays (because of nonce), but not MitM
can use public key server that A needs not to save B public key

asymmetric way (both ways)
n are nonces, m are sent messages, K is symmetric session key
use asymmetric to send nonces (na, nb)
nonces confirm key is established with correct associate
A → B m_1 = {na}_pB
B → A m_2 = {na, nb, K}_pA
A → B m_3 = {nb}_K

7.5.6 key agreement

with one time pads
A → B m_1 = {K}_a
B → A m_2 = {m_1}_b
A can now XOR with a and K, and therefore learns b
A → B m_3 = {K}_b
but advisory can learn K too if all messages known

with diffie hellman
choose public c and p
A → B m_1 = 5^a mod p
B → A m_2 = 5^b mod p
key = m_1^b mod p = m_2^a mod p
not safe against MitM

7.5.7 attacks

replays
simply resend messages without knowing exact content
use nonces which are only valid once
use increasing sequence numbers
use encrypted send time and max timeout at receiver

MitM
attacker redirects traffic between A and B to himself
use certificates / asymmetric cryptography

key faking
attacker sits between key server & A, A & B
can trick A into accepting a public key from attacker

7.6 interlock protocol

B → A sends challenge only A can answer
A → B sends encrypted answer, but only half of bits
A → B sends rest of the answer
B checks that first message is received in very short time
X needs whole A message to impersonate A
if X forwards first part immediately, X is not able to perform MitM
if X buffers till both messages received then B knows about intruder

7.7 authentication with certificates

certificates of A is singed by a trusted authority
A → B secret encrypted with public key of B
B → A sends back decrypted secret, confirming it has the private key

7.8 zero knowledge proof

A proofs knowledge to B without giving away the solution
verifier and prover interact together
but verifier can only prove to himself that prover knows answer

example graph isomorphy
prover says he knows isomorph graphs G1 = G2
prover construct H by renaming random knots of G1 or G2
verifier then requests mapping to G1 or G2
prover can do this easily as he knows H ~ G1 and G1 ~ G2
process is repeated