Distributed Systems

Network protocols & standards

OSI Network Layers

1: Physical Layer
- Service: Transmission of a raw bit stream over a communication channel
- Functions: Conversion of bits into electrical or optical signals
- Examples: X.21, Ethernet (cable, detectors & amplifiers)

2: Data Link Layer
- Service: Reliable transfer of frames over a link
- Functions: Synchronization, error correction, flow control
- Examples: HDLC (high level data link control protocol), LAP-B (link access procedure, balanced), LAP-D (link access procedure, D-channel), LLC (link level control), ...

3: Network Layer
- Service: Transfer of packets inside the network
- Functions: Routing, addressing, switching, congestion control
- Examples: IP, X.25

4: Transport Layer
- Service: Transfer of data between hosts
- Functions: Connection establishment, management, termination, flow-control, multiplexing, error detection
- Examples: TCP, UDP, ISO TP0-TP4

5: Session Layer
- Service: Coordination of the dialogue between application programs
- Functions: Session establishment, management, termination
- Examples: RPC
Distributed Systems
Network protocols & standards

6: Presentation Layer
- Service: Provision of platform independent coding and encryption
- Functions: Code conversion, encryption, virtual devices
- Examples: ISO code conversion, PGP encryption

7: Application Layer
- Service: Network access for application programs
- Functions: Application/Os specific
- Examples: APIs for mail, ftp, ssh, scp, discovery protocols ...

Serial Peripheral Interface (SPI)
- Used by gazillions of devices ... and it's not even a formal standard!
- Speed only limited by what both sides can survive!
- Usually push-pull drivers, i.e. fast and reliable, yet not friendly to wrong wiring/programming.
Network protocols & standards

- **Ethernet / IEEE 802.3**
  - Local area network (LAN) developed by Xerox in the 70s.
  - 10 Mbps specification by DEC, Intel, & Xerox in 1980.
  - First standard as IEEE 802.3 in 1985 (10 Mbps over thin-coaxial cables).
  - Currently using 1 Gbps (IEEE 802.3u) that uses copper and fiber optic cables in most desktops and laptops.
  - Current standards up to 100 Gbps (IEEE 802.3ba 2018).
  - More than 85% of current LAN lines worldwide according to the International Data Corporation (IDC).

  - Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

- **Ethernet / IEEE 802.11**
  - Wireless local area network (WLAN) developed in the 90s.
  - First standard as IEEE 802.11 in 1997 (1-2 Mbps over 2.4 GHz).
  - Typical usage at 54 Mbps over 2.4 GHz carrier at 20 MHz bandwidth.
  - Current standards up to 800 Mbps (IEEE 802.11ac) over 5 GHz carrier at 160 MHz bandwidth.
  - Future standards are designed for up to 100 Gbps over 60 GHz carrier.
  - Direct relation to IEEE 802.3 and similar OSI layer associations.

  - Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)
  - Direct-Sequence Spread Spectrum (DSSS)

- **Bluetooth**
  - Wireless local area network (WLAN) developed in the 90s with different features than 802.11.
  - Lower power consumption.
  - Shorter range.
  - Lower data rates (typically < 1 Mbps).
  - Ad-hoc networking (no infrastructure required).
  - Combinations of 802.11 and Bluetooth OSI layers are possible to achieve the required features set.

- **Token Ring / IEEE 802.5 / Fibre Distributed Data Interface (FDDI)**
  - "Token Ring" developed by IBM in the 70s.
  - IEEE 802.5 standard is modelled after the IBM Token Ring architecture (specifications are slightly different, but basically compatible).
  - IBM Token Ring requests are star topology as well as twisted pair cables, while IEEE 802.5 is uniplexing and time-domain multiloop, with a dual-ring, fiber-optical physical network.
  - Unlike CSMA/CD, Token ring is deterministic with respect to timing requirements.
  - FDDI is deterministic and failure resistant.

Uses: All types of devices currently used in performance oriented applications.
### Distributed Systems

#### Network protocols & standards

**Fibre Channel**

- Developed in the late 80s.
- ANSI standard since 1994.
- Current standards allow for 16 Gbps per link.
- Allows for three different topologies:
  - Point-to-point: 2 addresses
  - Arbitrated loop (similar to token ring): 127 addresses
  - Switched fabric: 2^n addresses, many topologies and concurrent data links possible
- Defines OSI's equivalent layers up to the session level.
- Mostly used in storage arrays, but applicable to super-computers and high integrity systems as well.

**FibreChannel IP (FC-IP)**

- Definied by the InfiniBand Trade Association (IBTA) since 1999.
- Allows for three different topologies:
  - Point-to-point: 2 addresses
  - Switched fabric: 2^n addresses, many topologies and concurrent data links possible
  - Switched fabric topologies. Concurrent data links possible (commonly up to 12 or 16 Gbps).
- Definies only the data-link layer and parts of the network layer.
- Existing devices use copper cables (instead of optical fibres).
- Mostly used in super-computers and clusters but applicable to storage arrays as well.

- Defines OSI's equivalent layers up to the session level.
- Cheaper than Ethernet or FibreChannel at high data-rates.
- Small packets (only up to 4 kB) and no session control.

### Distributed Systems

#### Distribution!

**Motivation**

Possibly...

- Fits an existing physical distribution (e-mail system, devices in a large craft, ...).
- High performance due to potentially high degree of parallel processing.
- High reliability/integrity due to redundancy of hardware and software.
- Scalable.
- Integration of heterogeneous devices.

Different specifications will lead to substantially different distributed designs.

### Distributed Systems

#### Some common phenomena in distributed systems

1. **Unpredictable delays** (communication)
   - Are we done yet?
2. **Missing or imprecise time-base**
   - Calculated relative or temporal relation!
3. **Partial failures**
   - Likelihood of individual failures increases
   - Likelihood of complete failure decreases (in case of a good design)

### Distributed Systems

#### Time in distributed systems

**Two alternative strategies:**

- **Based on a shared time** → Synchronize clocks!
- **Based on sequence of events** → Create a virtual time!

### Distributed Systems

#### Real-time clocks

- Discrete: i.e. time is not dense and there is a minimal granularity
- Affected

**Maximal clock drift:** Defined as:

\[
(1 - \delta) < \frac{C_i(t_i - t_j)}{C_j} < (1 + \delta)
\]

Often simplified as PPM (Parts-Per-Million)

Typical = 30PPM in computer applications.
**Distributed Systems**

**Synchronize a 'real-time' clock (bi-directional)**

Resetting the clock drift by regular reference time re-synchronization:

Maximal clock drift \( \delta \) defined as:

\[
(1 - \delta)^{-1} \leq \frac{C(b) - C(a)}{t_B - t_A} \leq (1 + \delta)
\]

'real-time' clock is adjusted forwards & backwards

\( t \) Calendar time

\( t \) 'measured time'

\( t \) clock

\( t \) clock

\( t \) clock

\( t \) clock

\( t \) 'measured time'

\( t \) clock

\( t \) clock

\( t \) 'measured time'

\( t \) clock

\( t \) Clock drifts affect fairness, but not integrity of the critical region.

**Distributed critical regions with synchronized clocks**

**Analysis**

- No deadlock, no individual starvation, no livelock.
- Minimal request delay: 2L.
- Minimal release delay: 1L.
- Communications requires 2 \((N-1)\) messages (can be significantly improved by employing broadcast mechanisms).
- Clock drifts affect fairness, but not integrity of the critical region.

**Virtual (logical) time [Lamport 1978]**

\[ a \rightarrow b \Rightarrow C(a) < C(b) \]

with \( a \rightarrow b \) being a causal relation between \( a \) and \( b \)

and \( C(a), C(b) \) are the virtual times associated with \( a \) and \( b \)

\( a \rightarrow b \) if:

- \( a \) happens earlier than \( b \) in the same sequential control-flow
- \( a \) denotes the sending event of message \( m \)
- \( b \) denotes the receiving event of the same message \( m \)
- there is a transitive causal relation between \( a \) and \( b \)

\[ a \rightarrow b \; \text{if} \; a \rightarrow b \; \text{and} \; b \rightarrow c \; \text{for some} \; c \]

Notion of concurrency:

\[ a \land b \Rightarrow \neg (a \land b) \land \neg (b \land a) \]

**Implications:**

- \( C(a) < C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)

**Virtual (logical) time**

\[ a \rightarrow b \Rightarrow C(a) < C(b) \]

Implications:

- \( C(a) < C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)

- \( C(a) --> C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)
- \( C(a) --> C(b) \Rightarrow ? \)


### Distributed Systems

#### Virtual (logical) time

\[ a \rightarrow b \Rightarrow C(a) < C(b) \]

Implications:

\[ C(a) < C(b) \Rightarrow \neg \left( b \rightarrow a \right) = \neg \left( b \land a \right) \]

\[ C(a) = C(b) \Rightarrow a \land b = \neg \left( a \lor b \right) \land \neg \left( b \lor a \right) \]

\[ C(a) < C(b) < C(c) = \neg \left( c \rightarrow a \right) = \neg \left( a \lor c \right) \land \left( a \lor c \right) \]

\[ C(a) < C(b) < C(c) = \neg \left( c \rightarrow a \right) = \neg \left( a \lor c \right) \land \left( a \lor c \right) \]

- Events in concurrent control flows are not ordered.
- No global order of time.

#### Distributed critical regions with logical clocks

- **Analysis**

  - No deadlock, no individual starvation, no livelock.
  - Minimal request delay: \( N - 1 \) requests (1 broadcast) + \( N - 1 \) replies.
  - Minimal release delay: \( N - 1 \) release messages (or 1 broadcast).
  - Communications requirements per request: \( 3(N - 1) \) messages (or \( N - 1 \) messages + 2 broadcasts).
  - Clocks are kept recent by the exchanged messages themselves.

  **Assumptions**

  1. No messages are lost
  2. Violations leads to stall

#### Distributed states

- **Elected a central coordinator (the Bully algorithm)**

  1. **Process \( P \) sends an Election-message to all processes with higher process numbers.**

  2. **Process \( P \) waits for response-message.**

  - If \( P \) does not receive a response-message after a predefined amount of time,
    - \( P \) declares itself the new coordinator and sends out a Coordinator-message to all.
  - If any \( P \) receives a Coordinator-message from \( P \), then \( P \) is the new coordinator.

- **How to read the current state of a distributed system?**

  This "god's eye view" does in fact not exist.
Distributed Systems

**Distributed states**

How to read the current state of a distributed system?

Instead: some entity probes and collects local states.

- What state of the global system has been accumulated?

- Connecting all the states to a global state.

**Snapshot algorithm**

Save \( s_i \) (as an individual token-message, or as part of another message):

- Observation process \( P_i \) (any process) creates a snapshot token \( s_i \) and saves its local state \( s_{i0} \).
- \( P_i \) sends \( s_i \) to all other processes.

Instead: some entity probes and collects local states.

- What state of the global system has been accumulated?

- Sorting the events into past and future events.

**Running the snapshot algorithm:**

- Observation process \( P_i \) (any process) creates a snapshot token \( s_i \) and saves its local state \( s_{i0} \).
- \( P_i \) sends \( s_i \) to all other processes.
- \( \forall P_j \) which received \( s_i \) (as an individual token-message, or as part of another message):
  - Save local state \( s_{i0} \) and send \( s_i \) to \( P_j \).
  - Attach \( s_i \) to all further messages, which are to be sent to other processes.
  - Save \( s_i \) and ignore all further incoming \( s_i \)’s.
- \( \forall P_j \) which previously received \( s_i \) and received a message without \( s_i \):
  - Rewind in to \( P_j \) (this message belongs to the snapshot).

A consistent global state (snapshot) is defined by a unique division into:

- "The Past" \( P \) (events before the snapshot):
  \( e \in P \) \( \Rightarrow \) \( e = e_{i0} \subseteq P \)

- "The Future" \( F \) (events after the snapshot):
  \( e \in F \) \( \Rightarrow \) \( e = e_{i0} \in F \)

Distributed Systems
Running the snapshot algorithm:

- \( P_i \), which received \( r_i \) (as an individual token message, or as part of another message):
  - Save local state \( x_i \) and send \( y_i \) to \( P_j \).
  - Attach \( y_i \) to all further messages, which are to be sent to other processes.
  - Save \( y_i \) and ignore all further incoming \( r_j \)’s.

Sorting the events into past and future events:

- Past and future events uniquely separated \( \rightarrow \) Consistent state.

A distributed server (load balancing)

- A ring of servers
- Clients:

Consistent distributed states

Why would we need that?

- Find deadlocks.
- Find termination/completion conditions.
- ... any other global safety of liveliness property.
- Collect a consistent system state for system backup/restore.
- Collect a consistent system state for further processing (e.g., distributed databases).
- ...
A distributed server (load balancing)

Task body PrintServer is
begin
  select
  accept Send_To_Server (Print_Job : in Job_Type; Job_Done : out Boolean)
  do
    if not Print_Job in Turned_Down_Jobs then
      if Not_Too_Busy then
        Applied_For_Jobs := Applied_For_Jobs + Print_Job;
        Next_Server.On_Ring.Contention (Print_Job; Server_Id);
      else
        null; -- removing the contention message free ring
        exit;
      end if;
      if Server_Id > Current_Task then
        Internal_Print_Server.Cancel_Print (Print_Job);
      else
        Turned_Down_Jobs := Turned_Down_Jobs + Print_Job;
      end if;
      exit;
    end if;
  end select;
end Print_Server;

Transactions

Atomic operations (spawning multiple processes)

How to ensure consistency in a distributed system?

Concurrency and distribution in systems with multiple, interdependent interactions?

Concurrent and distributed client/server interactions beyond single remote procedure calls?

Atomicity
- All or none of the sub-operations are performed.
  Atomicity helps achieve crash resilience. If a crash occurs, then it is possible to roll back the system to the state before the transaction was invoked.

Consistency
- Transforms the system from one consistent state to another consistent state.
  Consistency helps achieve a correct state. If a crash occurs, then it is possible to roll back the system to the state before the transaction was invoked.

Isolation
- Results (including partial results) are not revealed unless the transaction commits.
  Isolation ensures data serializability, so that all transactions appear to execute atomically. If the transaction commits, if the operation accesses a shared data object, the operation does not interfere with other operations on the same object.

Durability
- After a commit, results are guaranteed to persist, even after a subsequent system failure.
  Durability helps achieve crash resilience. Even if a crash occurs, the results of the committed transaction are guaranteed to persist.
For the \textit{serializability} of two transactions \textit{it is necessary and sufficient} for the order of their invocations of all conflicting pairs of operations to be the same for all the objects which are invoked by both transactions.

\begin{itemize}
  \item Two conflicting pairs of operations with the same order of execution.
  \item Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
  \item The order between processes also leads to a global order of processes.
\end{itemize}

\textit{Determining order in distributed systems requires logical clocks.}

\begin{itemize}
  \item Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
  \item The order between processes also leads to a global order of processes.
  \item Not serializable
\end{itemize}
Distributed Systems

Distributed Systems
Achieving serializability

For the serializability of two transactions it is necessary and sufficient for the order of their invocations of all conflicting pairs of operations to be the same for all the objects which are invoked by both transactions.

- Define Serialization graph: A directed graph; Vertices represent transactions; Edges $P_i \rightarrow P_j$ represent an established global order dependency between all conflicting pairs of operations of these two transactions.
- For the serializability of multiple transactions it is necessary and sufficient that the serialization graph is acyclic.

Distributed Systems

Transaction schedulers

Three major designs:
- Locking methods: Impose strict mutual exclusion on all critical sections.
- Time-stamp ordering: Note relative starting times and keep order dependencies consistent.
- "Optimistic" methods: Go ahead until a conflict is observed – then roll back.

Distributed Systems

Transaction schedulers – Locking methods

- Complete resource allocation before the start and release at the end of every transaction:
  - This will impose a strict sequential execution of all critical transactions.
- Observe two-phase locking:
  - Each transaction follows the following two-phase pattern during its operation:
    - Growing phase:
      - Locking phase: locks can be acquired, but not released.
      - Writing phase: locks can be released, but not acquired (two phase locking)
      - or locks are released on commit only (strict two phase locking):
      - Pessimistic methods: Impose strict mutual exclusion on all critical sections.
      - No-locking methods: Create a shadow copy of all the involved objects, perform all required operations on the shadow copy locally and finally commit the transactions.
      - "Optimistic" methods: Go ahead until a conflict is observed – then roll back.

Distributed Systems

Transaction schedulers – Optimistic control

Three sequential phases:
1. Read & execute:
   - Create a shadow copy of all involved objects and perform all required operations on the shadow copy and locally (i.e. in isolation).
2. Validate:
   - After local commit, check all occurred inter-views for serializability.
3. Update or abort:
   - If serializability could be ensured in step 2 then all results of involved transactions are written to all involved objects in dependency order of the transactions.
   - Otherwise destroy shadow copies and start over with the failed transactions.

Distributed Systems

Transaction schedulers – Optimistic control

How to create a consistent copy? (full isolation and maximal concurrency)

1. Read & execute:
   - Create a shadow copy of all involved objects and perform all required operations on the shadow copy and locally (i.e. in isolation).
2. Validate:
   - After local commit, check all occurred inter-views for serializability.
3. Update or abort:
   - If serializability could be ensured in step 2 then all results of involved transactions are written to all involved objects in dependency order of the transactions.
   - Otherwise destroy shadow copies and start over with the failed transactions.
   - How to implement "commit" and "abort" operations in a distributed environment?
**Distributed Systems**

**Two phase commit protocol**

**Start up (initialization) phase**

- **Data**
- **Ring of servers**

**Phase 1: Determine result state**

- Coordinator requests and assembles votes: "Commit" or "Abort"

**Phase 2: Implement results**

- Coordinator instructs everybody to "Commit"

**Ring of servers**

- **Client**
- **Data**
- **Operation**

**Determine coordinator**

**Setup & Start operations**

- **Coordinator**
- **Shadow copy**
- **Setup & Start operations**

- **Client**
- **Server**

- **Server**
- **Server**
- **Server**
- **Server**

**Everybody commits**
Two phase commit protocol

Phase 1: Client invokes a transaction

Phase 2: Implement results

- Everybody destroys shadows
- Everybody reports "Committed" or "Aborted"

Coordinator instructs everybody to "Abort"

Potential aborts along the way.

Side-aspect "data replication": large body of literature on this topic (see: distributed databases, operating systems, shared memory, cache management, ...)

Redundancy (replicated servers)

Premise:
A crashing server computer should not compromise the functionality of the system (full fault tolerance)

Assumptions & Means:
- A computer inside the server cluster might crash without losing functionality
- Replication: at least k + 1 servers
- The server cluster can recover any time (and specifically after the loss of a computer)
- Hot stand-by components, dynamic server group management
- The server is described fully by the current state and the sequence of messages received
- State machines we have to implement consistent state adjustments (re-organization) and consistent message passing (order needs to be preserved)

Start-up (initialization) phase

Job message received locally

Job message received by all server computers

Endurable

Processed

Received

Redundancy (replicated servers)
Redundancy (replicated servers)

Event: Server crash, new servers joining, or current servers leaving.

Server re-configuration is triggered by a message to all (this is assumed to be supported by the distributed operating system).

Each server on reception of a re-configuration message:
1. Wait for local job to complete or time-out.
2. Store local consistent state $S_i$.
3. Re-organize server ring, send local state around the ring.
4. If a state $S_j$ with $j > i$ is received then $S_i < S_j$.
5. Elect coordinator.
6. Enter 'Coordinator-' or 'Replicate-mode'.

Summary

- Networks
  - OSI, topologies
  - Practical network standards
- Time
  - Synchronized clocks, virtual (logical) times
  - Distributed critical regions (synchronized, logical, token ring)
- Distributed systems
  - Elections
  - Distributed states, consistent snapshots
  - Distributed servers (replicates, distributed processing, distributed commits)
  - Transactions (ACID properties, serializable interleavings, transaction schedulers)