References for this chapter

[Bacon1998]
Bacon, J
Concurrent Systems

[Ben2006]
Ben-Ari, M
Principles of Concurrent and Distributed Programming
second edition, Prentice-Hall 2006

[Schneider1990]
Schneider, Fred
Implementing fault-tolerant services using the state machine approach: a tutorial
ACM Computing Surveys 1990
vol. 22 (4) pp. 299-319

[Tanenbaum2001]
Tanenbaum, Andrew
Distributed Systems: Principles and Paradigms
Prentice Hall 2001

[Tanenbaum2003]
Tanenbaum, Andrew
Computer Networks
Prentice Hall, 2003
Network protocols & standards

**OSI network reference model**

Standardized as the **Open Systems Interconnection (OSI)** reference model by the International Standardization Organization (ISO) in 1977

- 7 layer architecture
- Connection oriented

Hardy implemented anywhere in full ...

...but its concepts and terminology are widely used, when describing existing and designing new protocols ...
Network protocols & standards

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)
Network protocols & standards

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
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
Network protocols & standards

7: Application Layer

- **Service**: Network access for application programs
- **Functions**: Application/OS specific
- **Examples**: APIs for mail, ftp, ssh, scp, discovery protocols …
Network protocols & standards

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.
**Serial Peripheral Interface (SPI)**

Full Duplex, 4-wire, flexible clock rate

- **Master**
  - Receive shift register
  - Transmit shift register
  - Clock generator
  - Slave selector

- **Slave**
  - Transmit shift register
  - Receive shift register
  - MISO
  - MOSI
  - SCK
  - NSS
  - CS

© 2018 Uwe R. Zimmer, The Australian National University
Network protocols & standards

Serial Peripheral Interface (SPI)

Clock phase and polarity need to be agreed upon
**Network protocols & standards (SPI)**

- **MOSI**: Master Out, Slave In
- **MISO**: Master In, Slave Out
- **SCK**: Synchronous Clock
- **NSS**: Slave Select
- **Rx**: Receive shift register
- **Tx**: Transmit shift register
- **CRC controller**
- **Communication controller**
- **CRCEN**: CRC Enable
- **CRCNEXT**: CRC Next
- **CRCL**: CRC Limit
- **Baud rate generator**
- **NSS logic**
- **Address and data bus**

From STM32L4x6 advanced ARM®-based 32-bit MCUs reference manual: Figure 420 on page 1291

© 2018 Uwe R. Zimmer, The Australian National University
**Network protocols & standards (SPI)**

- **Master**
  - Receive shift register
  - Transmit shift register
  - Clock generator
  - Slave selector

- **Slaves**
  1. **Slave 1**
     - MISO → MISO
     - MOSI → MOSI
     - SCK → SCK
     - NSS → CS

  2. **Slave 2**
     - MISO → MISO
     - MOSI → MOSI
     - SCK → SCK
     - NSS → CS

  3. **Slave 3**
     - MISO → MISO
     - MOSI → MOSI
     - SCK → SCK
     - NSS → CS

**Master**
- Receive shift register
- Transmit shift register
- Clock generator
- Slave selector

**Slave**
- Receive shift register
- Transmit shift register
- Slave selector

**Full duplex with 1 out of x slaves**
Network protocols & standards (SPI)

- Master
  - Receive shift register
  - Transmit shift register
  - Clock generator
  - Slave selector

- Slave 1
  - Transmit shift register
  - Receive shift register
  - SCK
  - Slave selector

- Slave 2
  - Transmit shift register
  - Receive shift register
  - SCK

- Slave 3
  - Transmit shift register
  - Receive shift register
  - SCK

Concurrent simplex with \( y \) out of \( x \) slaves
Network protocols & standards (SPI)

Concurrent daisy chaining with all slaves

© 2018 Uwe R. Zimmer, The Australian National University
Network protocols & standards

OSI

Application
Presentation
Session
Transport
Network
Data link
Physical

TCP/IP

Application
Transport
IP
Network
Physical

OSI

Application
Presentation
Session
Transport
Network
Data link
Physical

User data
Network protocols & standards

OSI

- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical

TCP/IP

- Application
- Transport
- IP
- Network
- Physical

AppleTalk

- AppleTalk Filing Protocol (AFP)
  - AT Data Stream Protocol
  - AT Session Protocol
  - Zone Info Protocol
  - Printer Access Protocol
  - Routing Table Maintenance Prot.
  - AT Update Based Routing Protocol
  - Name Binding Prot.
  - AT Transaction Protocol
  - AT Echo Protocol
  - Datagram Delivery Protocol (DDP)
  - AppleTalk Address Resolution Protocol (AARP)
  - EtherTalk Link Access Protocol
  - LocalTalk Link Access Protocol
  - TokenTalk Link Access Protocol
  - FDDITalk Link Access Protocol
  - IEEE 802.3
  - LocalTalk
  - Token Ring
  - FDDI
### Network protocols & standards

#### OSI

<table>
<thead>
<tr>
<th>Layer</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>AppleTalk Filing Protocol (AFP)</td>
</tr>
<tr>
<td>Presentation</td>
<td>AT Data Stream Protocol, AT Session Protocol, Zone Info Protocol, Printer Access Protocol</td>
</tr>
<tr>
<td>Session</td>
<td>AT Update Based Routing Protocol, Name Binding Protocol, AT Transaction Protocol, AT Echo Protocol</td>
</tr>
<tr>
<td>Transport</td>
<td>IP, Datagram Delivery Protocol (DDP), AppleTalk Address Resolution Protocol (AARP)</td>
</tr>
<tr>
<td>Data link</td>
<td>IEEE 802.3, LocalTalk, Token Ring IEEE 802.5, FDDI</td>
</tr>
<tr>
<td>Physical</td>
<td>LocalTalk, IEEE 802.3, LocalTalk, Token Ring IEEE 802.5, FDDI</td>
</tr>
</tbody>
</table>

---

© 2018 Uwe R. Zimmer, The Australian National University
Network protocols & standards

Ethernet / IEEE 802.3

Local area network (LAN) developed by Xerox in the 70’s

- 10 Mbps specification 1.0 by DEC, Intel, & Xerox in 1980.
- First standard as IEEE 802.3 in 1983 (10 Mbps over thick co-ax cables).
- Currently 1 Gbps (802.3ab) copper cable ports used in most desktops and laptops.
- Currently standards up to 100 Gbps (IEEE 802.3ba 2010).
- More than 85% of current LAN lines worldwide (according to the International Data Corporation (IDC)).

Carrier Sense Multiple Access with Collision Detection (CSMA/CD)
Network protocols & standards

Ethernet / IEEE 802.3

OSI relation: PHY, MAC, MAC-client
Network protocols & standards

Ethernet / IEEE 802.3

OSI relation: PHY, MAC, MAC-client

802.3 MAC
Physical medium-independent layer

MAC Client

PHY

MII
Physical medium-dependent layers

MDI

Link

Link media, signal encoding, and transmission rate

MII = Medium-independent interface
MDI = Medium-dependent interface - the link connector
Network protocols & standards

Ethernet / IEEE 802.11

Wireless local area network (WLAN) developed in the 90’s

- 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 780 Mbps (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 association.

- Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)
- Direct-Sequence Spread Spectrum (DSSS)
Wireless local area network (WLAN) developed in the 90’s with different features than 802.11:

- Lower power consumption.
- Shorter ranges.
- 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.
Network protocols & standards

Token Ring / IEEE 802.5 / Fibre Distributed Data Interface (FDDI)

- “Token Ring” developed by IBM in the 70’s
- 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 unspecified in topology and medium
- Fibre Distributed Data Interface combines a token ring architecture with a dual-ring, fibre-optical, physical network.

Unlike CSMA/CD, Token ring is deterministic (with respect to its timing behaviour)

FDDI is deterministic and failure resistant

None of the above is currently used in performance oriented applications.
Network protocols & standards

Fibre Channel

- Developed in the late 80’s.
- 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 deterministic, real-time capable
  - **Switched fabric**: $2^{24}$ addresses, many topologies and concurrent data links possible

- Defines OSI equivalent layers up to the session level.

- Mostly used in storage arrays, but applicable to super-computers and high integrity systems as well.
Network protocols & standards

Fibre Channel

Mapping of Fibre Channel to OSI layers:

- **FC-0 Physical**
- **FC-1 Data link**
- **FC-2 Network**
- **FC-3 Common service**
- **FC-4 Protocol mapping**

OSI layers:
- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical

FC/IP layers:
- Application
- Transport
- IP
- Network
- Physical

TCP/IP layers:
- Application
- Transport
- Network
- Data link
- Physical

User data flows from top to bottom.
Network protocols & standards

InfiniBand

- Developed in the late 90’s
- Defined by the InfiniBand Trade Association (IBTA) since 1999.
- Current standards allow for 25 Gbps per link.
- Switched fabric topologies.
- Concurrent data links possible (commonly up to 12 x 300 Gbps).
- Defines 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.
- 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.
What can be distributed?

- **State**: Common operations on distributed data
- **Function**: Distributed operations on central data
- **State & Function**: Client/server clusters
- **none of those**: Pure replication, redundancy
Common design criteria

- Achieve **De-coupling** / high degree of local autonomy
- **Cooperation** rather than central control
- Consider **Reliability**
- Consider **Scalability**
- Consider **Performance**
Some common phenomena in distributed systems

1. Unpredictable delays (communication)
   - Are we done yet?

2. Missing or imprecise time-base
   - Causal relation or temporal relation?

3. Partial failures
   - Likelihood of individual failures increases
   - Likelihood of complete failure decreases (in case of a good design)
Time in distributed systems

Two alternative strategies:

*Based on a shared time* ➔ Synchronize clocks!

*Based on sequence of events* ➔ Create a virtual time!
‘Real-time’ clocks

- **discrete** – i.e. time is *not* dense and there is a minimal granularity
- **drift affected**:

Maximal clock drift $\delta$ defined as:

$$(1 + \delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq (1 + \delta)$$

often specified as PPM (Parts-Per-Million)

(typical $\approx$ 20 PPM in computer applications)
Synchronize a ‘real-time’ clock (bi-directional)

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

Maximal clock drift $\delta$ defined as:

$$(1 + \delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq (1 + \delta)$$

‘real-time’ clock is adjusted forwards & backwards

✉ Calendar time
Synchronize a ‘real-time’ clock (forward only)

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

Maximal clock drift $\delta$ defined as:

$$(1 + \delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq 1$$

‘real-time’ clock is adjusted forwards only

‟Monotonic time“
Distributed critical regions with synchronized clocks

1. Create OwnRequest and attach current time-stamp.  
Add OwnRequest to local RequestQueue (ordered by time).  
Send OwnRequest to all processes.

2. Delay by $2L$ ($L$ being the time it takes for a message to reach all network nodes)

3. While Top (RequestQueue) $\neq$ OwnRequest: delay until new message

4. Enter and leave critical region

5. Send Release-message to all processes.
Distributed Systems

Distributed Systems

Distributed critical regions with synchronized clocks

Analysis

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

Assumptions:
- $L$ is known and constant $\implies$ violation leads to loss of mutual exclusion.
- No messages are lost $\implies$ violation leads to loss of mutual exclusion.
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 \( b \)

\[ a \rightarrow b \text{ iff:} \]

- \( a \) happens earlier than \( b \) in the same sequential control-flow or
- \( a \) denotes the sending event of message \( m \),
  while \( b \) denotes the receiving event of the same message \( m \) or
- there is a transitive causal relation between \( a \) and \( b \): \( a \rightarrow e_1 \rightarrow \ldots \rightarrow e_n \rightarrow b \)

Notion of concurrency:

\[ a \parallel b \Rightarrow \neg(a \rightarrow b) \land \neg(b \rightarrow a) \]
Virtual (logical) time

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

Implications:

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

\[ C(a) = C(b) \Rightarrow ? \]

\[ C(a) = C(b) < C(c) \Rightarrow ? \]

\[ C(a) < C(b) < C(c) \Rightarrow ? \]
Virtual (logical) time

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

Implications:

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

\[ C(a) = C(b) \Rightarrow a \parallel b \]

\[ C(a) = C(b) < C(c) \Rightarrow ? \]

\[ C(a) < C(b) < C(c) \Rightarrow ? \]
Virtual (logical) time

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

Implications:

\[ C(a) < C(b) \Rightarrow \neg (b \rightarrow a) = (a \rightarrow b) \lor (a \parallel b) \]

\[ C(a) = C(b) \Rightarrow a \parallel b = \neg (a \rightarrow b) \land \neg (b \rightarrow a) \]

\[ C(a) = C(b) < C(c) \Rightarrow ? \]

\[ C(a) < C(b) < C(c) \Rightarrow ? \]
Virtual (logical) time

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

Implications:

\[ C(a) < C(b) \Rightarrow \neg (b \to a) = (a \to b) \lor (a \parallel b) \]

\[ C(a) = C(b) \Rightarrow a \parallel b = \neg (a \to b) \land \neg (b \to a) \]

\[ C(a) = C(b) < C(c) \Rightarrow \neg (c \to a) \]

\[ C(a) < C(b) < C(c) \Rightarrow \neg (c \to a) \]
Virtual (logical) time

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

Implications:

\[ C(a) < C(b) \Rightarrow \neg (b \rightarrow a) = (a \rightarrow b) \lor (a \parallel b) \]

\[ C(a) = C(b) \Rightarrow a \parallel b = \neg (a \rightarrow b) \land \neg (b \rightarrow a) \]

\[ C(a) = C(b) < C(c) \Rightarrow \neg (c \rightarrow a) = (a \rightarrow c) \lor (a \parallel c) \]

\[ C(a) < C(b) < C(c) \Rightarrow \neg (c \rightarrow a) = (a \rightarrow c) \lor (a \parallel c) \]
Virtual (logical) time

Time as derived from causal relations:

Events in concurrent control flows are not ordered.

No global order of time.
Implementing a virtual (logical) time

1. \( \forall P_i: C_i = 0 \)

2. \( \forall P_i: \)
   - \( \forall \) local events: \( C_i = C_i + 1 \);
   - \( \forall \) send events: \( C_i = C_i + 1 \); Send (message, \( C_i \));
   - \( \forall \) receive events: Receive (message, \( C_m \)); \( C_i = \max(C_i, C_m) + 1 \);
Distributed critical regions with logical clocks

- ∀ times: ∀ received Requests:
  - Add to local RequestQueue (ordered by time)
  - Reply with Acknowledge or OwnRequest

- ∀ times: ∀ received Release messages:
  - Delete corresponding Requests in local RequestQueue

1. Create OwnRequest and attach current time-stamp.
   - Add OwnRequest to local RequestQueue (ordered by time).
   - Send OwnRequest to all processes.

2. Wait for Top (RequestQueue) = OwnRequest & no outstanding replies

3. Enter and leave critical region

4. Send Release-message to all processes.
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:
- No messages are lost  ❁ violation leads to stall.
Distributed critical regions with a token ring structure

1. **Organize** all processes in a logical or physical **ring** topology

2. **Send** one **token** message to one process

3. \( \forall \) times, \( \forall \) processes: **On receiving** the **token** message:
   1. If required the process **enters** and **leaves** a critical section (while holding the token).
   2. The **token** is **passed** along to the next process in the ring.

Assumptions:
- Token is not lost \( \forall \) violation leads to stall.

(a lost token can be recovered by a number of means – e.g. the ‘election’ scheme following)
Distributed critical regions with a central coordinator

A global, static, central coordinator

- Invalidates the idea of a distributed system
- Enables a very simple mutual exclusion scheme

Therefore:

- A global, central coordinator is employed in some systems … yet …
- … if it fails, a system to come up with a new coordinator is provided.
Electing a central coordinator (the Bully algorithm)

Any process $P$ which notices that the central coordinator is gone, performs:

1. **$P$ sends** an *Election*-message to all processes with *higher* process numbers.

2. **$P$ waits** for response messages.
   - If no one responds after a pre-defined amount of time: $P$ declares itself the new coordinator and sends out a *Coordinator*-message to all.
   - If any process responds, then the election activity for $P$ is over and $P$ waits for a *Coordinator*-message.

All processes $P_i$ perform at all times:

- **If $P_i$ receives** a *Election*-message from a process with a *lower* process number, it **responds** to the originating process and starts an election process itself (if not running already).
Distributed Systems

Distributed Systems

Distributed states

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

- “The Past” $P$ (events before the snapshot):
  
  $$(e_2 \in P) \land (e_1 \rightarrow e_2) \Rightarrow e_1 \in P$$

- “The Future” $F$ (events after the snapshot):
  
  $$(e_1 \in F) \land (e_1 \rightarrow e_2) \Rightarrow e_2 \in F$$
Distributed Systems

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?

Sorting the events into past and future events.
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?

Event in the past receives a message from the future!
Division not possible!  Snapshot inconsistent!
Distributed Systems

Snapshot algorithm

- Observer-process \( P_0 \) (any process) creates a snapshot token \( t_s \) and saves its local state \( s_0 \).
- \( P_0 \) sends \( t_s \) to all other processes.
- \( \forall P_i \) which receive \( t_s \) (as an individual token-message, or as part of another message):
  - Save local state \( s_i \) and send \( s_i \) to \( P_0 \).
  - Attach \( t_s \) to all further messages, which are to be sent to other processes.
  - Save \( t_s \) and ignore all further incoming \( t_s \)'s.

- \( \forall P_i \) which previously received \( t_s \) and receive a message \( m \) without \( t_s \):
  - Forward \( m \) to \( P_0 \) (this message belongs to the snapshot).
Running the snapshot algorithm:

- Observer-process $P_0$ (any process) creates a snapshot token $t_s$ and saves its local state $s_0$.
- $P_0$ sends $t_s$ to all other processes.
Running the snapshot algorithm:

- $\forall P_i$ which receive $t_s$ (as an individual token-message, or as part of another message):
  - Save local state $s_i$ and send $s_i$ to $P_0$.
  - Attach $t_s$ to all further messages, which are to be sent to other processes.
  - Save $t_s$ and ignore all further incoming $t_s$'s.
Running the snapshot algorithm:

- $\forall P_i$ which previously received $t_s$ and receive a message $m$ without $t_s$:
  - **Forward** $m$ to $P_0$ (this message belongs to the snapshot).
Distributed Systems

Distributed Systems

Distributed states

Running the snapshot algorithm:

- $\forall P_i$ which receive $t_s$ (as an individual token-message, or as part of another message):
  - Save local state $s_i$ and send $s_i$ to $P_0$.
  - Attach $t_s$ to all further messages, which are to be sent to other processes.
  - Save $t_s$ and ignore all further incoming $t_s$'s.
Running the snapshot algorithm:

- Save $t_s$ and ignore all further incoming $t_s$'s.
Running the snapshot algorithm:

- **Finalize** snapshot
Running the snapshot algorithm:

- Sorting the events into past and future events.
- Past and future events uniquely separated
- Consistent state
Either

- Make assumptions about the communication delays in the system.

or

- Count the sent and received messages for each process (include this in the local state) and keep track of outstanding messages in the observer process.
Consistent distributed states

Why would we need that?

- Find deadlocks.
- Find termination / completion conditions.
- … any other global safety of liveness 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)
A distributed server (load balancing)
A distributed server (load balancing)
Distributed Systems

A distributed server (load balancing)
Distributed Systems

A distributed server (load balancing)
A distributed server (load balancing)

with Ada.Task_Identification; use Ada.Task_Identification;

task type Print_Server is
  entry Send_To_Server (Print_Job : in Job_Type; Job.Done : out Boolean);
  entry Contention     (Print_Job : in Job_Type; Server_Id : in Task_Id);
end Print_Server;
A distributed server (load balancing)

task body Print_Server is
begin
loop
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, Current_Task);
    requeue Internal_Print_Server.Print_Job_Queue;
  else
    Turned_Down_Jobs := Turned_Down_Jobs + Print_Job;
  end if;
end if;
end Send_To_Server;
(...)

or

accept Contention (Print_Job : in Job_Type; Server_Id : in Task_Id) do
  if Print_Job in AppliedForJobs then
    if Server_Id = Current_Task then
      Internal_Print_Server.Start_Print (Print_Job);
    elsif Server_Id > Current_Task then
      Internal_Print_Server.Cancel_Print (Print_Job);
      Next_Server_On_Ring.Contention (Print_Job; Server_Id);
    else
      null; -- removing the contention message from ring
    end if;
  else
    Turned_Down_Jobs := Turned_Down_Jobs + Print_Job;
    Next_Server_On_Ring.Contention (Print_Job; Server_Id);
  end if;
end Contention;

or

terminate;
end select;
end loop;
end Print_Server;
Transactions

Concurrency and distribution in systems with multiple, interdependent interactions?

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

Definition (ACID properties):

- **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.

- **Isolation**: Results (including partial results) are **not revealed unless and until the transaction commits**. If the operation accesses a shared data object, invocation 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.
Transactions

Definition (ACID properties):

- **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.

- **Isolation**: Results (including partial results) are not revealed unless and until the transaction commits. If the operation accesses a shared data object, invocation 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.

Questions:

- What hardware do we need to assume?
- How to ensure consistency in a distributed system?
- Actual isolation and efficient concurrency?
- Actual isolation or the appearance of isolation?
- Shadow copies?
Distributed Systems

Transactions

A closer look inside transactions:

- **Transactions** consist of a sequence of **operations**.
- If two operations out of two transactions can be performed *in any order with the same final effect*, they are **commutative** and **not critical** for our purposes.
- **Idempotent** and **side-effect free** operations are by definition **commutative**.
- All non-commutative **operations** are considered **critical operations**.
- Two **critical operations** as part of two different transactions while affecting the same object are called a **conflicting pair of operations**.
A closer look at multiple transactions:

- Any sequential execution of multiple transactions will fulfil the ACID-properties, by definition of a single transaction.
- A concurrent execution (or ‘interleavings’) of multiple transactions might fulfil the ACID-properties.

If a specific concurrent execution can be shown to be equivalent to a specific sequential execution of the involved transactions then this specific interleaving is called ‘serializable’.

If a concurrent execution (‘interleaving’) ensures that no transaction ever encounters an inconsistent state then it is said to ensure the appearance of isolation.
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.

(Determining order in distributed systems requires logical clocks.)
• Two conflicting pairs of operations with the same order of execution.


Distributed Systems

Serializability

 Serializable
• Two conflicting pairs of operations with different orders of executions.

Not serializable.
• Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
• The order between processes also leads to a global order of processes.
Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes also leads to a global order of processes.

Serializable
Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes also leads to a global order of processes.

Serializable
Three conflicting pairs of operations with the same order of execution (pair-wise between processes).

The order between processes does no longer lead to a global order of processes.

Not serializable
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 $i$ represent transactions $T_i$;
  Edges $T_i \rightarrow T_j$ represent an established global order dependency between all conflicting pairs of operations of those two transactions.

For the **serializability** of multiple transactions it is **necessary and sufficient** that the serialization graph is **acyclic**.
Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).

Serialization graph is acyclic.

Serializable
Three conflicting pairs of operations with the same order of execution (pair-wise between processes).

Serialization graph is cyclic.

Not serializable
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.
Transaction schedulers – Locking methods

Locking methods include the possibility of deadlocks. Be careful from here on out …

- **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.

- **(Strict) two-phase locking**:
  Each transaction follows the following two phase pattern during its operation:
  - *Growing phase*: locks can be acquired, but not released.
  - *Shrinking phase*: locks can be released *anytime*, but not acquired (two phase locking) or locks are released *on commit only* (strict two phase locking).

  - Possible deadlocks
  - Serializable interleavings
  - Strict isolation (in case of strict two-phase locking)

- **Semantic locking**: Allow for separate read-only and write-locks
  - Higher level of concurrency (see also: use of functions in protected objects)
**Transaction schedulers – Time stamp ordering**

Add a unique time-stamp (any global order criterion) on every transaction upon start. Each involved object can inspect the time-stamps of all requesting transactions.

- Case 1: A transaction with a time-stamp *later* than all currently active transactions applies:
  - the request is accepted and the transaction can go ahead.
- Alternative case 1 (strict time-stamp ordering):
  - the request is delayed until the currently active earlier transaction has committed.
- Case 2: A transaction with a time-stamp *earlier* than all currently active transactions applies:
  - the request is not accepted and the applying transaction is to be aborted.

- Collision detection rather than collision avoidance
  - No isolation
  - Cascading aborts possible.
- Simple implementation, high degree of concurrency
  - also in a distributed environment, as long as a global event order (time) can be supplied.
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 interleavings for serializability.

3. **Update or abort**:  
   3a. 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.  
   3b. Otherwise: destroy shadow copies and start over with the failed transactions.
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 interleavings for **serializability**.

3. **Update or abort:**
   - **3a.** 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**.
   - **3b.** Otherwise: **destroy** shadow copies and **start over** with the failed transactions.

**Full isolation and maximal concurrency!**

**How to create a consistent copy?**

**How to update all objects consistently?**

**Aborts happen after everything has been committed locally.**
Distributed transaction schedulers

Three major designs:

- **Locking methods**: no aborts
  Impose strict mutual exclusion on all critical sections.

- **Time-stamp ordering**: potential aborts along the way
  Note relative starting times and keep order dependencies consistent.

- **“Optimistic” methods**: aborts or commits at the very end
  Go ahead until a conflict is observed – then roll back.

How to implement “commit” and “abort” operations in a distributed environment?
Two phase commit protocol

Start up (initialization) phase
Two phase commit protocol

Start up (initialization) phase
Two phase commit protocol

Start up (initialization) phase
Two phase commit protocol

Start up (initialization) phase
Two phase commit protocol

Start up (initialization) phase
Two phase commit protocol

Start up (initialization) phase
Two phase commit protocol

Phase 1: Determine result state

Coordinator requests and assembles votes: "Commit" or "Abort"
Two phase commit protocol

Phase 2: Implement results

Coordinator instructs everybody to "Commit"
Two phase commit protocol

Phase 2: Implement results
Two phase commit protocol

Phase 2: Implement results
Two phase commit protocol

Phase 2: Implement results

Everybody reports "Committed"
Two phase commit protocol
or Phase 2: Global roll back

Coordinator instructs everybody to "Abort"
Two phase commit protocol

or Phase 2: Global rollback
Two phase commit protocol

Phase 2: Report result of distributed transaction

Coordinator reports to client: "Committed" or "Aborted"
Distributed Systems

Distributed Systems

Distributed transaction schedulers

Evaluating the three major design methods in a distributed environment:

- **Locking methods**: No aborts.
  Large overheads; Deadlock detection/prevention required.

- **Time-stamp ordering**: Potential aborts along the way.
  Recommends itself for distributed applications, since decisions are taken locally and communication overhead is relatively small.

- **“Optimistic” methods**: Aborts or commits at the very end.
  Maximizes concurrency, but also data replication.

- Side-aspect “data replication”: large body of literature on this topic
  (see: distributed data-bases / 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:

- $k$ computers inside the server cluster might crash without losing functionality.
  - Replication: at least $k + 1$ servers.

- The server cluster can reorganize 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).

[Schneider1990]
Redundancy *(replicated servers)*

Stages of each server:

1. Job message received by all active servers
2. Job message received locally
3. Job processed locally
4. Delivered
**Redundancy (replicated servers)**

Start-up (initialization) phase

![Diagram of a ring of identical servers connected in a circular network]
Redundancy (replicated servers)

Start-up (initialization) phase

Determine coordinator
Redundancy (replicated servers)

Start-up (initialization) phase

Coordinator determined
Redundancy (replicated servers)

Coordinator receives job message
Redundancy (replicated servers)

Distribute job

Coordinator sends job both ways
Redundancy (replicated servers)

Distribute job

Everybody received job (but nobody knows that)
Redundancy (replicated servers)

Processing starts

First server detects two job-messages and processes job.
Redundancy (replicated servers)

Everybody (besides coordinator) processes
Redundancy (replicated servers)

Coordinator processes

Coordinator also received two messages and processes job
Redundancy (replicated servers)

Result delivery

Coordinator delivers his local result
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 \( S_i \leftarrow S_j \)
5. Elect coordinator
6. Enter ‘Coordinator-’ or ‘Replicate-mode’
Summary

Distributed Systems

• 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)