Embedded Information Systems

Power-aware systems

6. Real-time communication

November 17, 2020

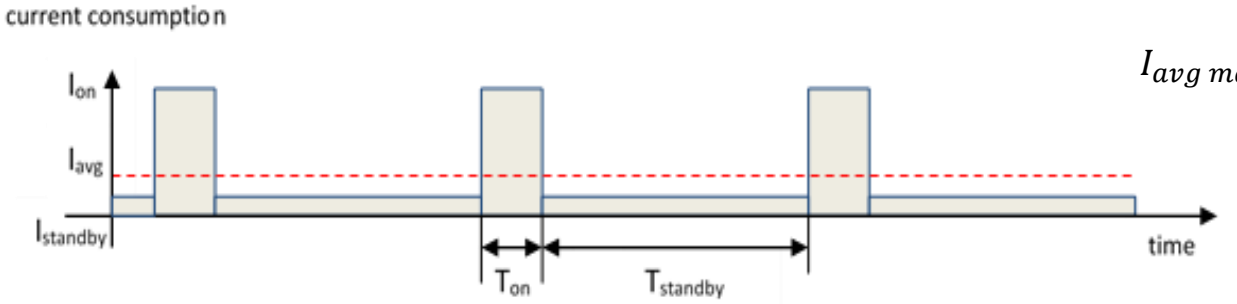
Reminder:

Power Aware Systems – Low Power Design

Example: The capacity/capability of two AA (Mignon) type battery cells in sensor network applications (microcontroller+radio+sensors):

2 pieces of AA battery cells have an average capacity of **3000 mAh.**

How long can we use our system in a day, if we require a total availability of services for minimum **1 year** (8760 hours), while **P_{on}=150 mW** (I_{on}=50mA) and **$I_{standby}$=50µA**?



$$I_{avg\ max} = \frac{3000mAh}{8760h} = 342\mu A$$

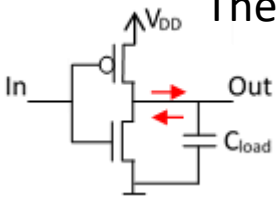
$$I_{avg} = \frac{I_{on}T_{on}}{T_{on}+T_{standby}} + \frac{I_{standby}T_{standby}}{T_{on}+T_{standby}} = I_{on}\lambda + I_{standby}(1-\lambda)$$

$$\lambda = \frac{I_{avg\ max} - I_{standby}}{I_{on} - I_{standby}} = 0.0058 \approx 0.6\%$$

$$\approx 8\frac{minutes}{day}.$$

If we take measurements **in every hour**, then they can take (only) **20 seconds**!

Power Consumption of a CMOS Gate:



The two FETs are alternately open and closed. Main sources of power consumption:

(1) charging and discharging capacitors,

(2) short circuit path between supply rails during switching,

(3) leaking diodes and transistors (becomes one of the major factors due to shrinking feature sizes in semiconductor technology).

Reminder:

Power consumption of CMOS circuits (ignoring leakage): $P \sim \alpha C_L V_{DD}^2 f$,

where V_{DD}: stands for **power supply**, α: **switching activity** (for a clock it is 1), C_L: a **load capacity**, f: **clock frequency**. Delay for CMOS circuits: $\tau \sim C_L \frac{V_{DD}}{(V_{DD} - V_T)^2}$ where V_T: threshold voltage, $V_T \ll V_{DD}$.

It can be stated: Decreasing V_{DD}
- reduces P **quadratically** (f constant);
- The gate delay increases only **reciprocally**;
- Maximal frequency f_{max} decreases **linearly.**

Potential for Energy Optimization (Dynamic Voltage Scaling: DVS): $P \sim \alpha C_L V_{DD}^2 f$,

$E \sim \alpha C_L V_{DD}^2 f t = \alpha C_{Lt} V_{DD}^2 (\#cycles)$.

Saving energy for a given task:
- reduce the supply voltage V_{DD};
- reduce switching activity;
- reduce the load capacitance;
- reduce the number of cycles (#cycles).

Use of Parallelism:

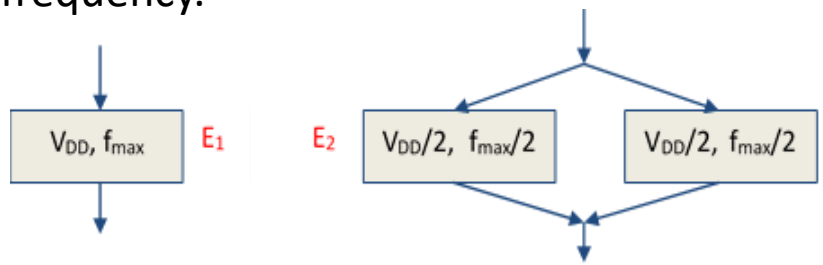
Duplicated hardware with half of the supply voltage, and half of the clock frequency.

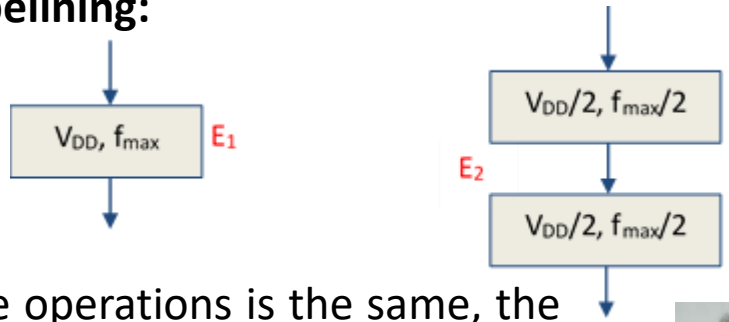
Power Supply Gating:

It is one of the most effective ways of minimizing static power consumption (leakage).

Power Supply Gating cuts-off power supply to inactive units/components: a header switch provides virtual power, while a footer switch provides virtual ground and thus reduces leakage.

Use of Pipelining:



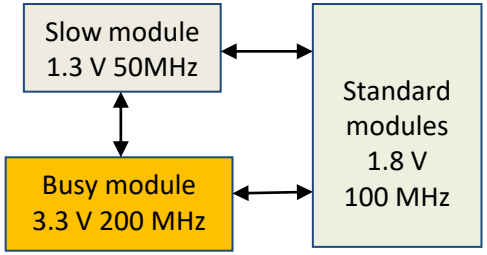


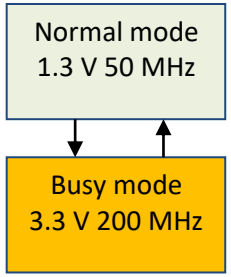
$E_1 \sim V_{DD}^2 (\#cycles)$, $E_2 = \frac{E_1}{4}$. The number of the operations is the same, the energy consumption is lowered to its fourth.

Reminder:

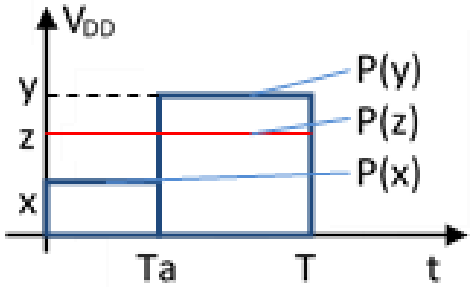
The application of **Very Long Instruction Word** (VLIW) architectures is an alternative **parallel instruction sets** are applied.

Not all components require same performance. Required performance may change over time:





Optimal strategy (Dynamic Voltage Scaling):



Case A: execute at voltage x for Ta time units, and at voltage y for $(1\text{-}a)T$ time units.
The energy consumption: $T(aP(x) + (1-a)P(y))$.
Case B: execute at voltage $z = ax + (1-a)y$ for T time units.
The energy consumption: $TP(z)$.

Since the power is a convex (quadratic) function of V_{DD}, therefore $P(z) < aP(x) + (1-a)P(y)$, i.e. it worth executing at constant voltage. (The linear combination gives a string above the parabola.)

Example:

V_{DD} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

Task execution needs 10^9 cycles within **25 seconds.**

a. Complete task **ASAP**: 10^9 cycles @ 50 MHz.
Energy consumption: $E_a = 10^9 * 40 * 10^{-9} = 40$ [J],
time requirement: $10^9 * 20 * 10^{-9} = 20s$.

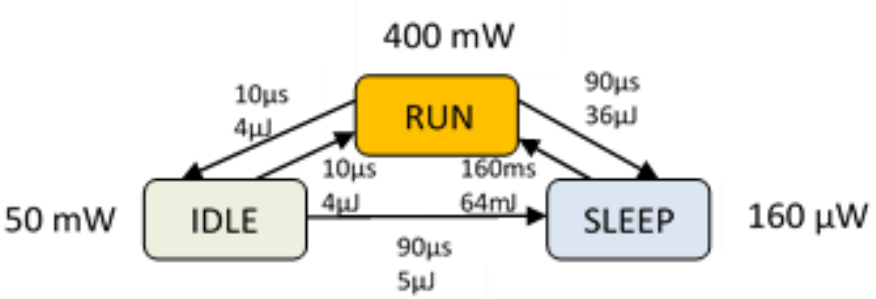
b.Execution **at two voltages**: 0.75*10^9 cycles @ 50 MHz + 0.25*10^9 cycles @ 25 MHz.
Energy consumption: $E_a = 0.75 * 10^9 * 40 * 10^{-9} + 0.25 * 10^9 * 10 * 10^{-9} = 32.5$ [J],
time requirement: $0.75 * 10^9 * 20 * 10^{-9} + 0.25 * 10^9 * 40 * 10^{-9} = \mathbf{25s}$.

Reminder:

c. Execution **at optimal voltage**: 10^9 cycles @ 40 MHz.
Energy consumption: $E_a = 10^9 * 25 * 10^{-9} = \mathbf{25\ [J]}$,
time requirement: $10^9 * 25 * 10^{-9} = \mathbf{25s.}$
Comment: Obviously some spare-time is always required at task executions.

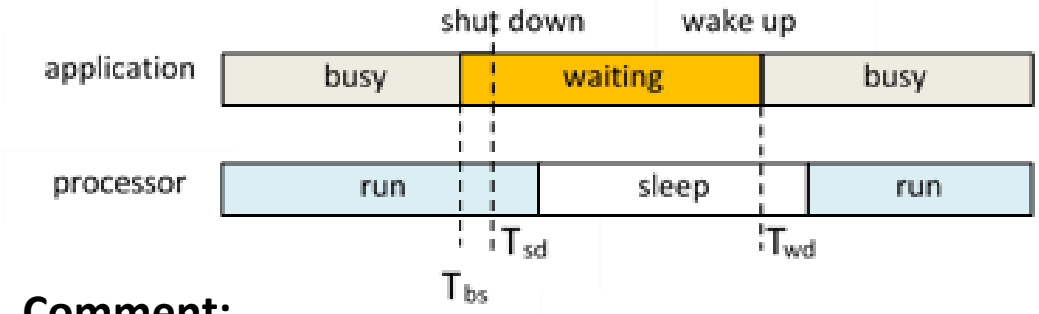
V_{DD} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

Dynamic Power Management (DPM): tries to assign optimal power saving states



Requires hardware support.
Example: **StrongARM SA1100**
IDLE: a software routine may stop the CPU when not in use, while monitoring interrupts.
SLEEP: shuts down on-chip activities.

Example: reduce power according to workload:



T_{bs}: time before shutdown
T_{sd}: shutdown delay
T_{wd}: wakeup delay

Comment:
Shutdown only if long idle times occur.
There is a trade-off between savings and overhead "costs".

Reminder:

Example: Dynamic power management:

Suppose that the power consumption $P(f)$ of a given CMOS processor at frequency f is:

$$P(f) = \left[10\left(\frac{f}{100MHz}\right)^3 + 20\right] mWatt$$

To reduce the power consumption, one can **adjust** the **execution frequency**.

The maximum/minimum available frequency:

$f_{max} = 1000MH\,z / f_{min} = 50MHz$

Frequency switching has **negligible overhead** and the processor can operate **at any frequency** between $50MHz$ and $1000MHz$.

One can also apply dynamic power management to turn the processor to the **sleep mode** (or turn the processor off) **to reduce** the power consumption.

When the processor is in the sleep mode, it consumes **no power**.

However, turning the processor on to the run mode requires **additional energy consumption**, i.e. $3 \times 10^{-5} Joule$. (Switching from run mode to sleep mode consumes no energy.)

Turning on/off the processor **can be done instantly**. The system has **three jobs** to execute:

	arrival time	deadline	execution cycles
τ_1	0	2ms	100000
τ_2	2ms	6ms	100000
τ_3	6ms	7ms	80000

The processor is in the run mode at time 0 and is required to be in the run mode at time 7 ms.

Problem#1: The energy consumption to execute *C* cycles is $\frac{CP(f)}{f}$.

There is a **critical frequency** f_{crit} between $50MHz$ and $1000MHz$ at which the energy consumption to execute any *C* cycles **is minimized.**

What is the critical frequency of the processor?

Reminder:

Solution#1: As we lower the frequency the power consumed falls.
But beyond a critical frequency, the rate of fall in power is overweighed by the fall in frequency and thus the power consumed per cycle increases.

To compute this critical frequency, we have to minimize $\frac{P(f)}{f}$.

$$P(f) = \left[10\left(\frac{f}{100MHz}\right)^3 + 20\right] mWatt$$

Let f be normalized to $100MHz$. The first derivative of $\frac{P(f)}{f}$ is $20f - \frac{20}{f^2}$, which equals 0 if $f = 1$.

Thus $f_{crit} = 100MHz$.

Problem#2: When the processor is idle at frequency f_{min} for t seconds, then the energy consumption is $P(f_{min}) \times t$. The **break-even time** is defined as the minimum idle interval for which it worth turning the processor off. What is **the break-even time** of the processor?

Solution#2: Going into the sleep mode must provide sufficient energy saving to compensate for the additional energy consumption (overhead) of $3 \times 10^{-5} + 0\, Joule$.

Energy(idle state, f_{min}) – Energy(sleep state) ≥ Energy(turning from sleep to run state)

$$P(f_{min}) \times t_{bev} - 0 \geq 3 \times 10^{-5} Joule$$

$$t_{bev} \geq \frac{3 \times 10^{-5} Joule}{10^{-3} \times (10 \times 0.5^3 + 20) Watt} = 1.412ms.$$

Problem#3:

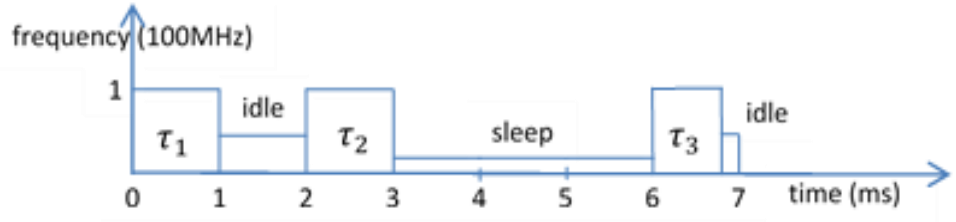
A *workload-conserving* schedule is defined as a schedule which always executes some jobs when the ready queue is empty. Provide the workload conserving schedule for the 3 tasks which minimizes the energy consumption without violating the timing constraints.
For this apply the *critical frequency* f_{crit} as the frequency for active task execution.

Reminder:

What is the energy consumption of the schedule?

Solution#3:

Since $P(f_{crit}) = 30mWatt$,
and $P(f_{min}) = 21.25mWatt$,



thus the energy consumption of the schedule is

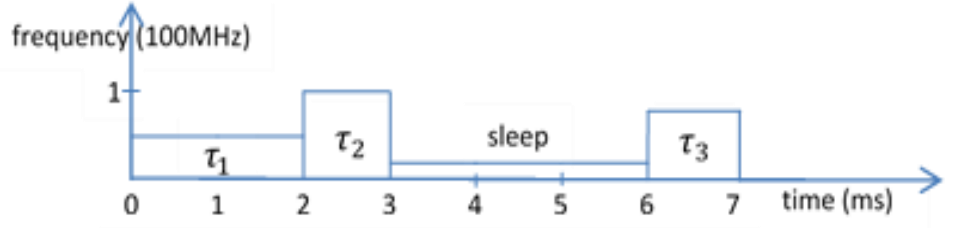
$$(\underbrace{30 \times 1}_{\tau_1} + \underbrace{21.25 \times 1}_{\text{idle}} + \underbrace{30 \times 1}_{\tau_2} + \underbrace{3 \times 10^1}_{\text{turning on to run}} + \underbrace{30 \times 0.8}_{\tau_3} + \underbrace{21.25 \times 0.2}_{\text{idle}})\mu Joule = 0.1395\ mJoule.$$

Note that the interval [1,2] is **shorter than the break/even time**, therefore there is no reason to switch to sleep mode.

Problem#4: Would it be possible to provide another **workload-conserving schedule** without violating the timing constraints for the 3 tasks, and **with less energy consumption**?

Solution#4: Yes, the solution is to use **the convex nature of the power consumption**: to slow down execution of tasks τ_1 and τ_3 to such extent as to avoid idle times after their executions. Thus, even though we work below the critical frequency, **we can save on energy consumption**.

The energy consumption of the schedule is (see figure):



$$(\underbrace{21.25 \times 2}_{\tau_1} + \underbrace{30 \times 1}_{\tau_2} + \underbrace{3 \times 10^1}_{\text{turning on to run}} + \underbrace{(10 \times 0.8^3 + 20) \times 1}_{\tau_3})\mu Joule = 0.12762mJoule.$$

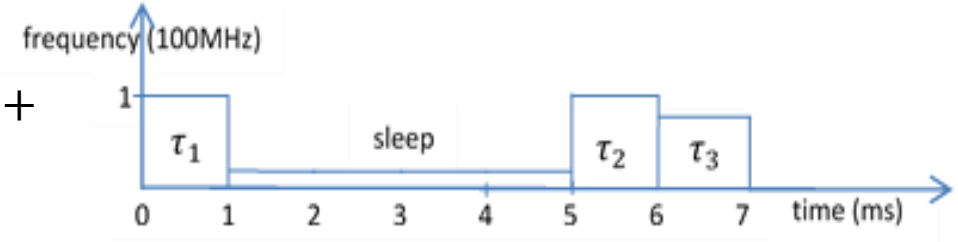
Reminder:

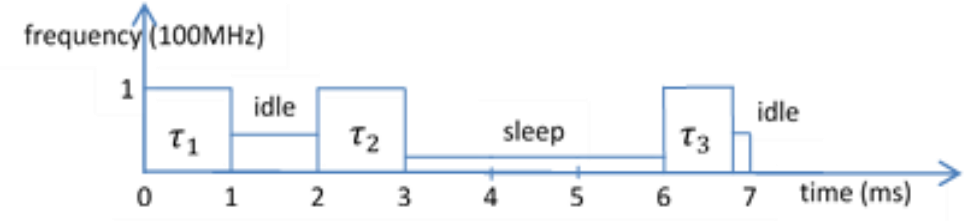
Problem#5: Would it be possible another schedule **without violating the timing constraints** for the 3 tasks that is not workload-conserving but the energy consumption is **even lower** than the optimal work/load conserving schedule?

Solution#5: Yes, the idea here is to ***batch* the sleep mode** into one block and execute τ_1 with critical frequency. This illustrates that to conserve energy; **workload-conserving** strategies are **not necessarily the best**.

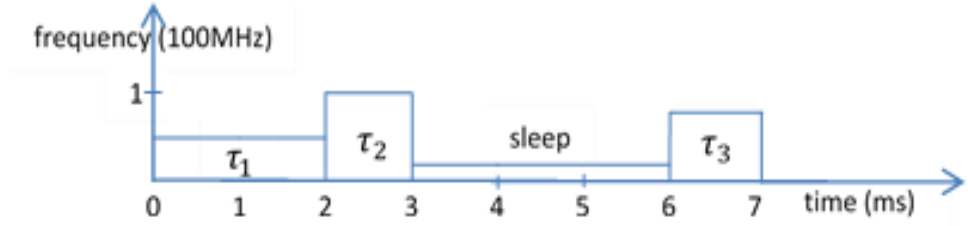
The energy consumption of the schedule is (see figure):

$(30 \times 1 + 3 \times 10^1 + 30 \times 1 + (10 \times 0.8^3 + 20) \times 1)\mu Joule = 0.115120 mJoule.$

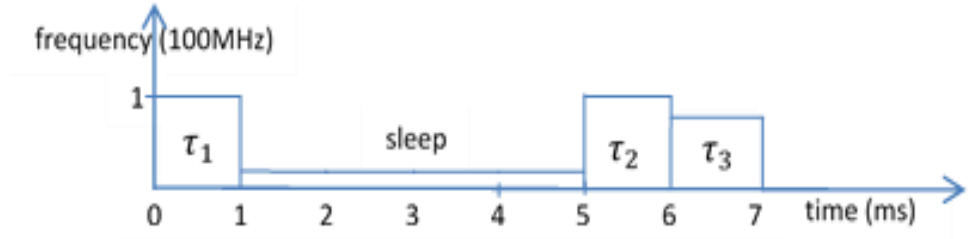




$0.1395\ mJoule$



$0.12762 mJoule$



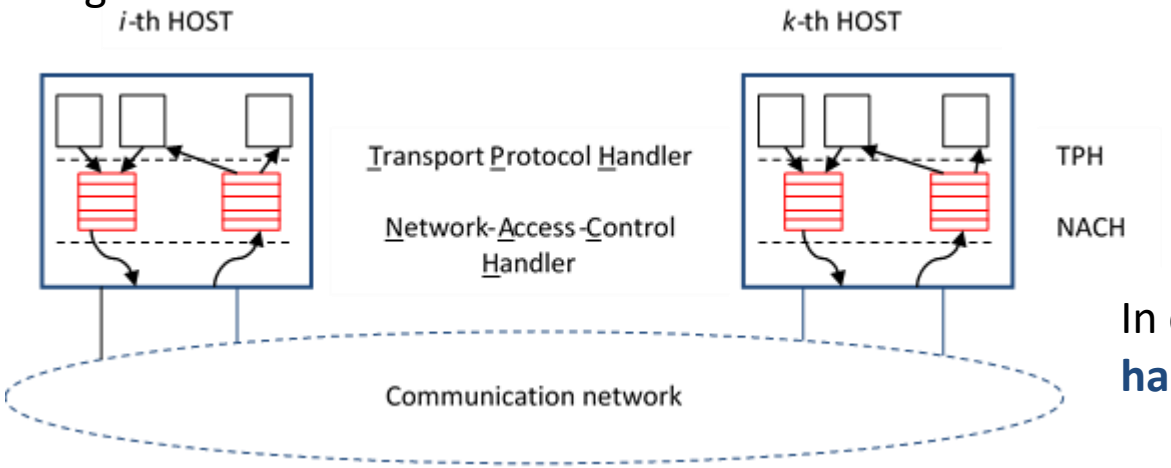
$0.115120 mJoule$

6. Real-time (RT) communication

In general, **complicated mechanisms,** different queues. The RT requirements are **hard to meet.**

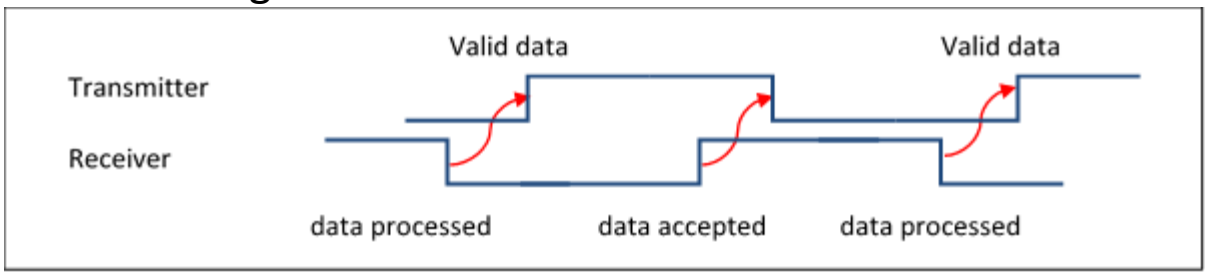
The criticality of time conditions can be identified even on the physical level.

The general scheme:



In case of asynchronous communication **handshaking is unavoidable**.

Handshaking with two wires:



The **speed** and time conditions of the asynchronous communication are determined by **both actors,** since till the processing of the received data, the transmitter cannot forward the next data.

Requirements:

1. A RT communication protocol should have a **predictable, and small maximum protocol latency and a minimal jitter.** The standard communication topology in distributed real-time systems is **multicast**, not point-to-point. A message should be delivered to all receivers within a short and known time interval.

2. Support for Composability:(1) Temporal encapsulation of the nodes:

the communication system should erect a **temporal firewall** around the operation of the host, **forbidding the exchange of control signals** across the Communication Network Interface (**CNI**). Thus, the communication system becomes **autonomous** and can be implemented and validated **independently** of the application software in the host.

(2) Fulfilling the obligations of the Client: a host implementing server functions **can guarantee its deadlines** if the clients fulfill their obligations, and do not overload the host with too many, uncoordinated service requests.

3. Flexibility: An RT protocol should be **flexible to accommodate different system configurations** without requiring a software modification and retesting of the operational nodes that are not affected by the change. As an example, imagine a car **with and without extras.**

4. Error detection: The communication system must provide predictable and dependable services. Errors must **be detected and corrected** without increasing the jitter of the protocol latency. If the errors cannot be corrected, **the receivers should be informed** about the error with low latency. Loss of information is of particular concern.

Consider a node, at a **control valve**, that receives output commands from another node. In case the communication is interrupted because the wires are cut, the control valve node should enter a safe state autonomously, e.g. it should close the valve.

The communication system must inform the control valve node about the loss of communication with low error detection latency. **End-to-end protocols are needed**.

(**Three Mile Island Nuclear Reactor #2** accident on March 28, 1978).

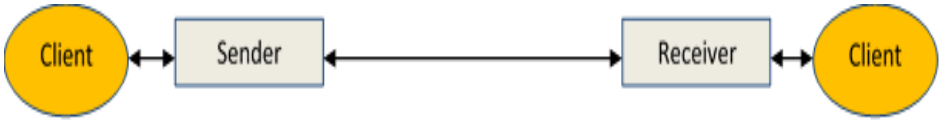
5. Physical structure: point-to-point communication can easily result in high costs.
Physical networks should be based on a bus or a ring structure.

Flow control:

Explicite flow control: Example: PAR (Positive Acknowledgement or Retransmission) protocol:

Many variants of the basic PAR protocol are known, but they all rely on the following principle:

(1) The **client** at the sender's site **initiates** the communication.
(2) The receiver has the authority to **delay the sender** via the bi-directional comm. channel.
(3) The communication error is detected **by the sender**, and not by the receiver.
The receiver is not informed when a communication error has been detected.
(4) **Time redundancy is used** to correct a communication error, thereby increasing the protocol latency in case of errors.



Program of the sender:

(1) The sender initializes a **retry counter to zero**.
(2) The sender **starts** a local time-out interval.
(3) The sender **sends** the message to the receiver.
(4) The sender **receives** an acknowledgement message from the receiver within the specified time-out interval.
(5) The sender **informs** its client about the successful transmission, and duly terminates.

If the sender does not receive a **positive acknowledgement** message from the receiver within the specified time-out interval:

(a) The sender **checks the retry counter** to determine whether the given maximum number of retries has already been exhausted.
(b) If so, **the sender aborts** the communication, and informs its client about the failure.
(c) If not, the sender **increments the retry counter** by one, and returns to (2).

Program of the receiver:

(1) If new message arrives at the receiver, **the receiver checks** whether this message has already been received.

(2) If not, the receiver **sends an acknowledgement** message to the sender, and **delivers** the message to its client.

(3) If yes, it just **sends another acknowledgement message** back to the sender.
(In this case the previous acknowledgement message has arrived at the sender out of the specified time-out interval or failed to arrive).

Example: Consider a bus system where a **token protocol controls media access** to the bus. The maximum **token rotation time** (TRT) is $10\ ms$.The time needed **to transport the message** on the bus is 1 ms. The length of **time-out interval**: 10+1+10+1=22 *ms*, since in worst-case the sender should wait 10 *ms* for the token, the message takes further 1 *ms,* and on the way back the worst case is the same.

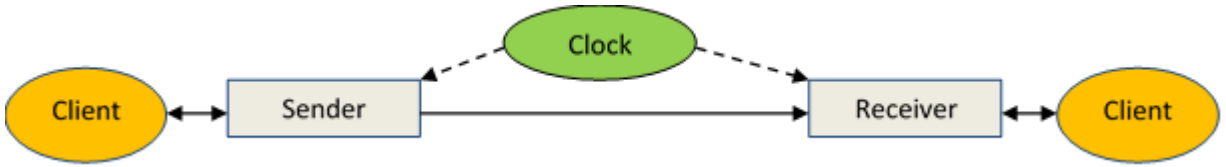
Here $d_{min} = \mathbf{1}\ ms$, a $d_{max} = (number\ of\ retries) * timeout + 10\ ms + 1\ ms$.
If the number of retries is two (i.e. we apply three trials), then d_{max}= **55** *ms*.

The value of some important features:

- **jitter**= $d_{max} - d_{min} = 54\ ms$.
- **The action delay,** if we have global clock: $d_{max} = 55\ ms$. If the granularity of the global clock is $100\ \mu s$, then the message becomes permanent after $d_{max} + 2g = 55.2\mu s$.
- **The action delay,** if the global clock is not available:: $2 * d_{max} - d_{min} = 109\ ms$.
- **The error detection latency**: 3*time-out: 66 *ms*.

This example illustrates that the explicit flow control in RT applications **might be disadvantageous** due to the large jitter and error detection latency.

Implicite flow control:



The communication is **time-triggered.** Both the sender and the receiver have a message scheduling timetable, which were fixed in design-time. From this schedule it is clear at what time the message is to be sent and received. **The sender** at the corresponding clock tick **pushes the message**, while at the same time **the receiver pulls the message** (push-pull mechanism). This approach fits better to the real-time requirements in many cases. E.g. **error detection** by the receiver is immediately possible, if the **expected message fails to arrive**.(For the sender this means a so-called fail-silent mode, where the absence of the message means the error state of the sender.)

Global time-base is needed. The sender transmits only **at fixed time instants**, no handshaking is applied, error detection is the role of the receiver, since it knows when a message should arrive. **Fault tolerance** is solved by **active redundancy**: *k* copies of the message are transmitted, and the transmission is successful unless at least one message arrives.

Summary of implicit flow control:

(1) The communication is initiated by clock tick.
(2) The receiver is expecting the message following the clock tick.
(3) Error is detected by the receiver, typically by realizing the absence of the message.
(4) For error correction active/hardware redundancy is applied.

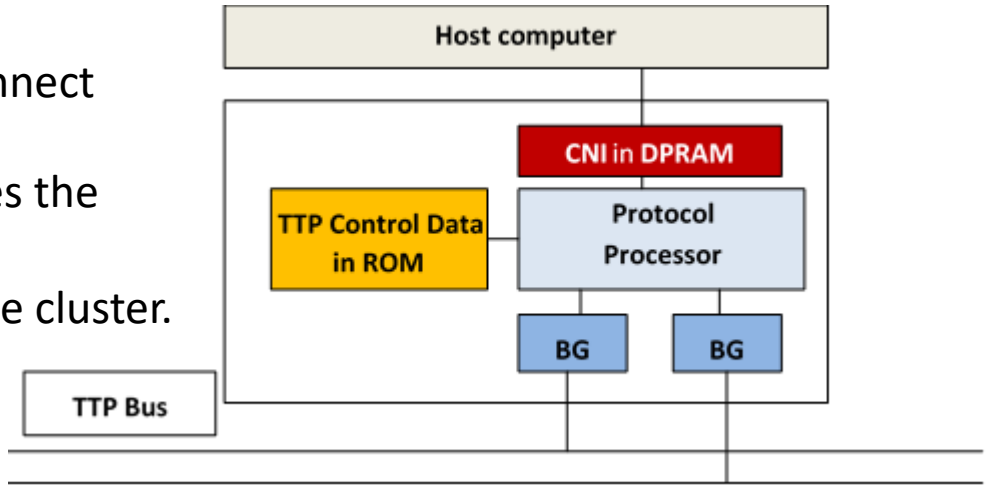
The Time Triggered Architecture (TTA) and the Time-Triggered Protocols (TTPs)

A detailed description can be found in:

HERMANN KOPETZ, GÜNTHER BAUER: *The Time-Triggered Architecture,* PROCEEDINGS OF THE IEEE, VOL. 91, NO. 1, JANUARY 2003, pp. 112-126.

Here follows only a brief description of TTA and TTP: It is devoted to implement **hard real-time** (HRT) **systems.** The basic building block of the TTA is a **node**.

A **node** comprises in a self-contained unit (possibly on a single silicon die) a **processor** with **memory**, **an input–output subsystem**, a TT **communication controller**,an operating system, and the relevant application software.



Two **replicated communication channels** connect the nodes, thus forming a **cluster.**

The **cluster** communication system comprises the physical interconnection network and the communication controllers of all nodes of the cluster.

In the TTA, the **communication system** is **autonomous** and executes periodically an a priori–specified **time-division multiple access** (TDMA) schedule.

It reads a state message from the **Communication Network Interface** (CNI) at the sending node at the a priori–known fetch instant and delivers it to the CNIs of all other nodes of the cluster at the a priori–known delivery instant, replacing the previous version of the state message.

The times of the periodic fetch and delivery actions are contained in the message scheduling table [the **message descriptor list** (MEDL)] of each communication controller.

It has two versions:

the TTP/C, which serves fault tolerant solutions,

the TTP/A, which is suitable for cheap field bus applications.

Each node consists of a **Host computer** and a **Communication Controller** (CC).

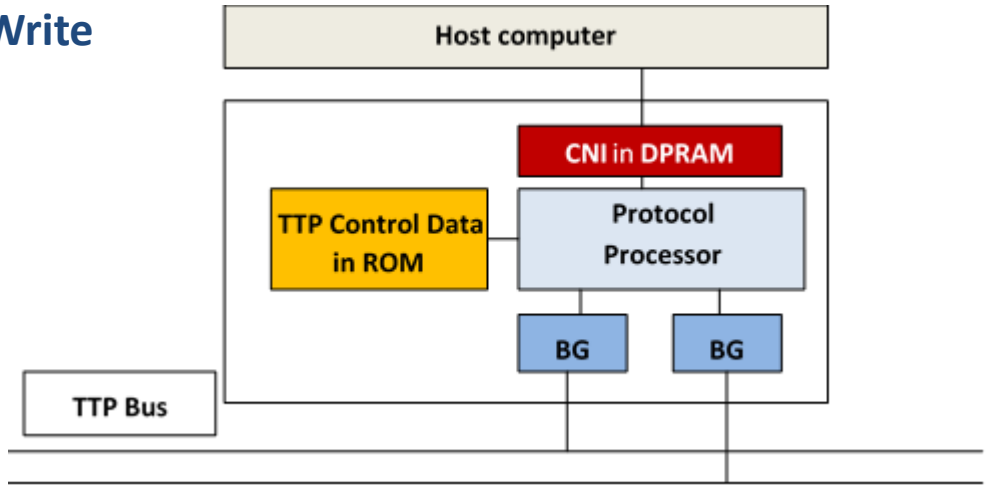
The CNI is the interface within node between the host and the CC.

It is a dual-port RAM (DPRAM).

Data integrity is solved by the **Non-Blocking Write** (NBW) **Protocol** (see later).

The local memory of the CC contains the **Message Description List** (MEDL), that determines which node can send, and which can receive a message at a given time.

The size of the MEDL is determined by the cluster round.



The TTP controllers contain - as independent hardware so-called **Bus Guardian** units, which monitor the **bus access patterns** of the controlling bus, and stop the operation of the controller, if the timing of the regular access patterns fails.

Important properties:

(1) The TTP is a **time-division-multiple-access** (TDMA) protocol.

(2) The CC is **autonomous**, which is controlled by the **MEDL** and the **global clock**.
This serves composability. The error of the hosts cannot influence the communication system, because **control signal cannot get through the CNI**, and **neither the MEDL can be accessed** from the host.

(3) The communication system is decided **in design time** (it is like a timetable at the railways): it knows in advance when a message arrives, and when a message is to be sent.
If the message fails to arrive, the **error is immediately detected**.

(4) **Naming:** the name of the message and the sender should not be part of the message, it can be read from the MEDL. At the same time, we can give different names to a given RT variable within the software of different hosts.

(5) **Acknowledgement:** We know in advance that all correct receivers receive the message of the correct sender.

As one of the receivers acknowledges the message, it can be assumed that all correctly operating host has also received it.

(6) **Fail-silence in the time domain:** TTP assumes that the nodes support the "fail silence" abstraction in the time domain, which means that the node either sends a message at right time, or sends nothing. This property within the TTP controller is solved by the **bus guardian.** The error handling in the magnitude domain is the responsibility of the host. The TTP provides only CRC.

The basic CNI:

The **CNI** is the most important interface within a time-triggered architecture because **it is the only interface** of the communication system that is **visible** to the software of the **host computer.** It thus constitutes the programming interface of a TTP network.

Status Registers	Control Registers
(S1) Global Internal Time	(C1) Watchdog
(S2) Node Time	(C2) Timeout Register
(S3) Message Description List	(C3) Mode Change Request
(S4) Membership	(C4) Reconfiguration Request
(S5) Status Information	(C5) External Rate Correction

The **Status Registers** are written by the TTP controller, while the **Control Registers** by the host.

S1: The common clock of the cluster on two bytes.

S2: The clock of node.

S3: MEDL Pointer.

S4: As many bits as the number of participants within the cluster.

If one of the bits is "TRUE", then that node was in operation within the last time-slot.

Status Registers	Control Registers
(S1) Global Internal Time	(C1) Watchdog
(S2) Node Time	(C2) Timeout Register
(S3) Message Description List	(C3) Mode Change Request
(S4) Membership	(C4) Reconfiguration Request
(S5) Status Information	(C5) External Rate Correction

C1: The host periodically restarts; the controller checks it.
If the restart fails, then the controller – supposing error – stops sending messages.
C2: Written by the host. If the time is over, it will cause interrupt.
For example, later, the host can synchronize its clock to that of the cluster.
C3: Using this a new scheduling can be introduced.
C4: In case of error a reconfiguration can be initiated.
C5: Makes possible external clock synchronization.

The Message Description List (MEDL)

Node Time	Address	D	L	I	A
When	*What: message pointer*	*direction*	*length*		

I: specifies whether the message is an initialization message or a normal message.

A: contains additional protective information concerning mode changes and mode role changes.

Fault-Tolerant Units: Its role is to mask the failure of a node. If it implements fail-silent abstraction, then it is enough to duplicate the nodes to tolerate a single value failure.
If the node does not implement the fail-silent abstraction, and can have value-failure at the CNI, then Triple Modular Redundancy (TMR) is to be applied.

If in case of node failure, nothing is known about the behaviour of the node, then byzantine error might also occur, i.e. four nodes can mask the error.

Fundamental conflicts in protocol design

A balanced protocol design tries to reconcile many requirements. It is important to understand which requirements are compatible with each other, and which requirements are in fundamental conflict with each other, and cannot be reconciled by any design decisions that are made.

Conflict: External control ↔ Composability

Consider a distributed real-time system consisting of a set of nodes that communicate with each other. Each node has a host computer with CNI. Composability in the temporal domain requires that:

- The CNI of every node is fully specified in the temporal domain;
- The integration of a set of nodes into the complete system does not led to any change of the temporal properties of the individual CNIs, and
- The temporal properties of every host can be tested in isolation with respect to the CNI.

If the temporal properties are not contained in the CNI specification, e.g. because the moment when a message must be transmitted is external and unknown to the communication system, then it is not possible to achieve composability in the temporal domain. If the temporal properties of the CNI are fully specified, then low-level composability can be achieved. There is, however, always the possibility that the application functions interact in an unpredictable manner that precludes high-level composability.

In an event-triggered system, the temporal signals originate external to the communication system, in the hosts of the nodes. It is thus not possible to achieve low-level composability.

Example: If all the nodes can compete at any point in time for a single communication channel on a demand basis, then, it is impossible to avoid the side effects caused by the extra transmission delay resulting from conflicts regarding the access to this single channel, no matter how clever the medium access protocol may be. These extra transmission delays can invalidate the temporal accuracy of the real-time images that are transported in the message.

Conflict: Flexibility ↔ Error detection

Flexibility implies that the behaviour of a node is not restricted a priori.
In an architecture without replication, error detection is only possible if the actual behaviour of the node can be compared to some a priori knowledge of the expected behaviour.
If such knowledge is not available, it is not possible to protect the network from a faulty node.

Example: Consider an event-triggered system with no regularity assumptions, where access to a single bus is determined solely by the message priority: if there is no restriction on the rate at which a node may send messages, it is impossible to avoid the monopolization of the network by a single (possibly erroneous) node that sends a continuous sequence of messages of the highest priority.

Example: If a node is not required to send a "heartbeat message" at regular intervals, it is not possible to detect a node failure with a bounded latency.

Conflict: Sporadic data ↔ Periodic data

A RT protocol can be effective in either the transmission of periodic data or the transmission of sporadic data, but not with both. The transmission of periodic data (e.g. data exchanges needed to coordinate a set of control loops) must take place with minimal latency jitter.
Because the repetitive intervals between the transmissions of periodic data are known a priori, conflict-free schedules can be designed in design time.

Sporadic data must be transmitted with minimal delay, on demand, at a priori unknown points in time. If an external event requiring the transmission of a sporadic message occurs at the same time as the next transmission of the periodic data, then, the protocol must decide either to delay the sporadic data, or to modify the schedules of the periodic data.

In either case, the latency jitter increases: one cannot satisfy both goals simultaneously.

Conflict: Single locus of control ↔ Fault tolerance

Any protocol that relies on a single locus of control has a single point of failure.
This is evident for a communication protocol that relies on a central master.
However, even the access method of token passing relies on a single locus of control at any particular moment, with no consideration of time as the control element.
If the station holding the token fails, no further communication is possible until the token loss has been detected by an additional time-out mechanism, and the token has been recovered.
This takes time, and also interrupts the real-time communication.
In some respects, the nontrivial problem of token recovery is related to the problem of switching from a central master to a standby master in a multi-master protocol.

Conflict: Probabilistic access ↔ Replica determinism

Another fundamental conflict exists between the property of replica determinism (needed if active redundancy is to be applied) and that of medium access based on probabilistic mechanisms. If systems that rely on a single winner emerging from fine-grained race conditions (e.g. bit arbitration, conflict resolution based on random numbers), it cannot be guaranteed that the access to the replicated communication channels is always resolved identically by competing nodes. Without replica determinism, each replica can come to different correct result, thereby leading to inconsistency in the system as a whole.

Performance Limits in TT systems

As in any distributed computing system, the performance of the TTA depends primarily on the available **communication bandwidth** and **computational power**.
Because of physical effects of time distribution and limits in the implementation of the guardians, a minimum interframe gap of about 5 µs must be maintained between frames to guarantee the correct operation of the guardians. If a bandwidth utilization of about 80% is intended, then the message-send phase must be in the order of about 20 µs, implying that about 40 000 messages can be sent per second within such a cluster.

With these parameters, a **sampling period** of about 250 µs can be supported in a cluster comprising **ten nodes.**

The amount of data that can be transported in the 20 µs window depends on the bandwidth:

If the bandwidth is 5Mbit/s then $5*10^{6}*20*10^{-6}$ = 100 bit (~12 byte) can be forwarded.

If the bandwidth is 1Gbit/s then $1*10^{9}*20*10^{-6}$ = 20 000 bit (2500 byte) can be forwarded.

Synchronizing ET and TT systems

The processor of the host operates in **ET mode**, while the network in **TT mode**.
This means that the CNI cannot be blocked without consequences.
The writing from the network is investigated.

Non-blocking Write Protocol (NBW):

At the interface there is one writer, the communication system, and many readers, the tasks of the host. A reader does not destroy the information written by the writer, but the writer can interfere with the operation of the reader.

In the NBW protocol, the writer is never blocked.

It will thus write a new version of the message into the DPRAM of the CNI whenever a new message arrives.

If a reader reads the message while the writer is writing a new version, the retrieved message will contain inconsistent information and must be discarded.

If the reader is able to detect the interference, then the reader can retry the read operation until it retrieves a consistent version of the message.

It must be shown that the number of retries performed by the reader is bounded.

The protocol requires a concurrency control field, CCF, for every message written.

Atomic access to the CCF must be guaranteed by the hardware.

The CCF is initialized to zero and incremented by the writer before start of the write operation.

It is again incremented after the completion of the write operation.

The reader starts by reading the CCF at the start of the read operation.

If the CCF is odd, then the reader retries immediately because a write operation is in progress.

At the end of the read operation the reader checks whether the CCF has been changed by the writer during the read operation.

If so, it retries the read operation again until it can read an uncorrupted version of the data structure.

It can be shown that upper bound for the number of read retries exists if the time between write operation is significantly longer than the duration of a write or read operation.

Initialization: CCF:=0

Writer:

```
Start:   CCF_old:=CCF;
         CCF:=CCF_old+1;
         <write into data srtucture>
         CCF:=CCF_old+2;
```

Reader:

```
Start:   CCF_begin:=CCF;
         if CCF_begin=odd then goto Start;
         <read data structure>
         CCF_end:=CCF;
         if CCF_end ≠ CCF_begin then goto Start;
```


Characteristics of a Communication Channel:

A communication channel is characterized by **its bandwidth** and its **propagation delay**.

Bandwidth: The bandwidth indicates the number of bits that can traverse a channel in unit time.

It is determined by the physical characteristics of the channel.

In a harsh environment, such as a car, it is not possible to transmit more than 10kbit/sec over a single-wire channel or 1 Mbit/sec over an unshielded twisted pair because of EMI constraints. In contrast, optical channels can transport gigabits of data per second.

Propagation Delay: The propagation delay is the time interval it takes for a bit to travel from one end of the channel to the other end. It is determined by the length of the channel and the transmission speed of the wave (electromagnetic, optical) within the channel.

The **transmission speed** of an electromagnetic wave in vacuum is about 300 000 km/sec, or 1 foot/nsec. Because the transmission speed of a wave in a cable is approximately 2/3 of the transmission speed of light in vacuum, it takes a signal about 5 μs to travel across a cable of 1 km length.

The term ***bit length*** of a channel is used to denote the number of bits that can traverse the channel within one propagation delay.

For example, if the channel bandwidth is 100 Mbit and the channel is 200 m long, the bit length of the channel is 100 bits, since the propagation delay of this channel is 1 μs.

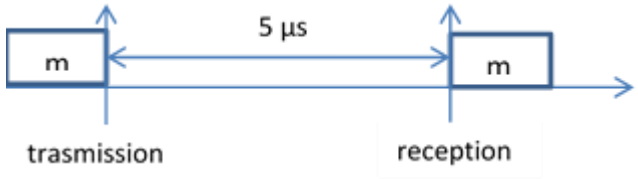
Limit to Protocol Efficiency: In a bus system, the data efficiency of any media access protocol to a single channel is limited by the need of to maintain a minimum time interval of one propagation delay between two successive messages.

Assume the bit length of a channel to be bl bits and the message length to be m bits.

Then an upper bound for the data efficiency of any media access protocol in a bus system is given by:

$$data\ efficiency < \frac{m}{m + bl}$$

Example: Consider a 1 km bus with a bandwidth equal to 100 Mbits/sec. The message length that is transmitted over the channel is 100 bits. It follows that the bit length of the channel is 500 bits, and the limit to the data efficiency is $100/(500 + 100) = 16.6\%$.



Properties of Transmission Codes:

The terms asynchronous and synchronous have different meanings depending on whether they are used in the computer-science community or in the data-communication community. In asynchronous communication, the receiver synchronizes its receiving logic with that of the sender only at the beginning of a new message.

Since the clocks of the receiver and the sender drift apart during the interval of the message reception, the message length is limited in asynchronous communication,

e.g. to about 10 bits in a UART (Universal Asynchronous Receiver Transmitter) device that uses a low-cost resonator with a drift rate of $10^{-2} sec/sec$.

In synchronous communication, the receiver resynchronizes its receive logic during the reception of a message to the ticks of the sender's clock. This is only possible if the selected data encoding guarantees frequent transitions in the bit stream.

A code that supports the resynchronization of the receiver's logic to the clock of the sender during transmission is called synchronizing code.

NRZ code: (non-return-to-zero)***:*** non-synchronizing code

11010001: "1" corresponds to high level, "0" correspond to low level

Manchester code: synchronizing code

11010001: "1" corresponds to rising edge: from low to high, "0" corresponds to a falling edge: from high to low. These edges appear in the middle of the clock interval. Always the next bit tells whether the signal should return to the other level at time instant of the clock, or not. If the new bit equals the previous one, then it should return.

This code is ideal from the point of view of resynchronization, but it has the disadvantage that the size of a feature element, i.e. the smallest geometric element in the transmission sequence, is half of the bit cell.

Modified Frequency Modulation Code (MFM):

The MFM code is a code that has a feature size of one-bit cell and is also synchronizing. The encoding scheme requires distinguishing between a data point and a clock point.

A "0" is encoded by no signal change at data point;

a "1" requires a signal change at data point.

If there are more than two "0"s in sequence, the encoding rules require a signal change at clock points.

Time Synchronization in Wireless Sensor Networks

Classes of Synchronization:

- ***Internal versus external***

The synchronization of all clocks in the network to a time supplied from outside the network is referred to as **external synchronization**. NTP performs external synchronization, and so do sensor nodes synchronizing their clocks to a master node.
Internal synchronization is the synchronization of all clocks in the network, without a predetermined master time. The only goal here is the consistency among the network nodes.

- ***Lifetime: Continuous versus on-demand***

The **lifetime of synchronization** is the period of time during which synchronization is required to hold. If time synchronization is continuous, the network nodes strive to maintain synchronization (of a given quality) at all times. For some sensor-network applications, **on-demand synchronization** can be as good as continuous synchronization in terms of synchronization quality, but much more efficient. During the (possibly long) periods of time between events, no synchronization is needed, and communication and hence energy consumption can be kept at a minimum. As the time intervals between successive events become shorter, a break-even point is reached where continuous and on-demand synchronization perform equally well. There are two kinds of on-demand synchronizations:

Event-triggered* on-demand synchronization** is based on the idea that to timestamp a sensor event, a sensor needs a synchronized clock only immediately after the event has occurred. It can then compute the timestamp for the moment in the recent past when the event occurred (Post-facto synchronization***).

***Time-triggered* synchronization** is used if we are interested in obtaining sensor data from multiple sensor nodes for a specific time. This means that there is no event that triggers the sensor nodes, but the nodes must take a sample at precisely the right time.

This can be achieved via ***immediate*** synchronization (where sensor nodes receive the order to immediately take a sample and time-stamp it) or ***anticipated*** synchronization (where the order is to take the sample at some future time, the ***target time***). Anticipated synchronization is necessary if it cannot be guaranteed that the order can be transmitted rapidly and simultaneously to all involved sensor nodes. This is especially the case if sensor nodes are more than one hop away from the node giving the order.

Analogously to the event-triggered ***post-facto synchronization***, we might refer to time-triggered synchronization as ***pre-facto synchronization***.

- ***Scope: all nodes versus subsets***

The *scope* of synchronization defines which nodes in the network are required to be synchronized. Depending on the application, the scope comprises all or only a subset of the nodes (where and when synchronization is required). Event-triggered synchronization can be limited to collocated subset of nodes which observe the event in question.

- ***Rate* synchronization *versus offset* synchronization**

Rate synchronization means that nodes measure identical time-interval lengths.

In a scenario where sensor nodes measure the time between the appearance and disappearance of an object, rate synchronization is a sufficient and necessary condition for comparing the duration of the object's presence within the sensor range of different nodes.

Offset synchronization means that nodes measure identical points in time, that is at some time *t*, the software clocks of all nodes in the scope show *t*. ***Offset synchronization*** is needed for combining timestamps from different nodes.

- ***Timescale transformation versus clock synchronization***

Time synchronization can be achieved in two fundamentally different ways.

We can synchronize clocks, that is make all clocks display the same time at any given moment. To achieve this, we must perform **rate and offset synchronization** (or continuous offset synchronization, which however is costly in terms of energy and bandwidth and requires reliable communication links).

The other approach **is to transform timescales**, that is to transform local times of one node into local times of another node.

The approaches in that **clock synchronization** requires either communication across the whole network or some degree of global coordination.

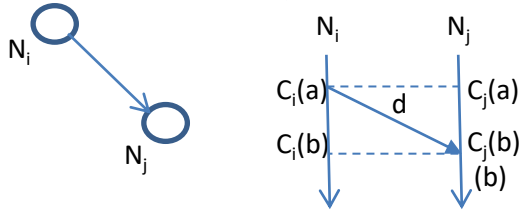
Timescale transformation does not have this drawback, but instead requires additional computations and memory overhead, since the received timestamps must be transformed.

- ***Time instants versus time intervals***

Time information can be given by specifying time instants (e.g., "t=5") or time intervals ("t∈ $[4.5,5.5]$"). In both cases, the time information can be refined by adding a statement of quality. E.g., the time information may be guaranteed to be correct with a certain probability, or even probability distributions can be given. Typically, the term *time uncertainty* is used.
In sensor networks the use of guaranteed time intervals can be very attractive.

Synchronization techniques: Taking one sample

Unidirectional Synchronization:



Node N_j is not familiar with d, its knowledge is only the fact, that the clock of node N_i displayed the value $C_i(a)$ before the clock of node N_j displayed $C_j(b)$.

To perform synchronization, we must estimate either the value of $C_j(a)$ or $C_i(b)$.

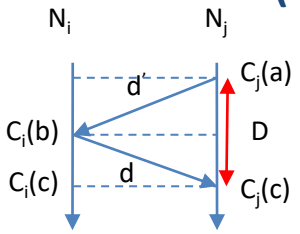
If the limits $d_{min} \leq d \leq d_{max}$ are known, then

$$\hat{C}_j(a) \approx C_j(b) - \frac{d_{min} + d_{max}}{2}$$ or $$\hat{C}_i(b) \approx C_i(a) + \frac{d_{min} + d_{max}}{2}$$

Having these estimates, the clock of node N_j should be modified by $\hat{C}_j(a) - C_i(a)$ or $C_j(b) - \hat{C}_i(b)$.

If the communication jitter $(d_{max} - d_{min})$ is large, then the synchronization will be inaccurate, because the lower bound of $C_j(a)$ will be $C_j(b) - d_{max}$, and the upper bound will be $C_j(b) - d_{min}$, which is a wide range.

Bidirectional (round trip) synchronization:



Here node N_j knows that $0 \leq d \leq D$. $D = C_j(c) - C_j(a)$. If $d_{min} \leq d \leq d_{max}$, then $\max(D - d_{max}, d_{min})$ and $\min(d_{max}, D - d_{min})$ give the limits of d.

The estimate that can be computed here:

$$\hat{C}_j(b) \approx C_j(c) - \frac{D}{2},$$

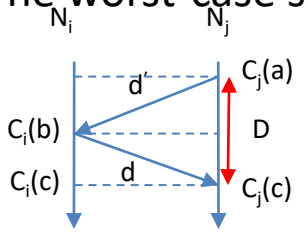
having lower bound $C_j(c) - (D - d_{min})$, and upper bound $C_j(c) - d_{min}$.

The clock of node N_j is to be modified by

$$\hat{C}_j(b) - C_i(b).$$

With such a method the quality of the synchronization will be better.

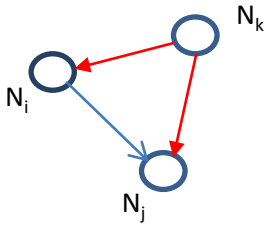
The worst-case synchronization error: $\frac{D}{2} - d_{min}$, that can be proved using the figure.

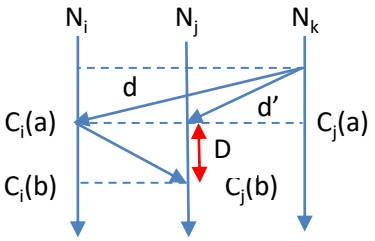


The method can be improved using the so-called probabilistic time synchronization, where node N_j after receiving the timestamp checks weather the value of $\frac{D}{2} - d_{min}$ < than a defined threshold. If not, then the request will be repeated.

Reference broadcasting synchronization:

In this case also a so-called *beacon* node N_k is involved. The beacon sends a broadcast message to the other nodes.





The delays are almost equal:

$d \approx d'.$ Thus $\hat{C}_i(b) \approx C_i(a) + D,$

i.e., the clock of node N_j should be modified by $C_j(b) - \hat{C}_i(b)$.

It is an important property, that the synchronization of node N_j is performed without using it radio channel.

The accuracy degrades with the hop distance from the root!

Synchronization of multiple nodes:

(1) Single-hop synchronization with a set of master nodes which are synchronized e.g. using GPS;

(2) Partitioning the network into clusters: all nodes within a cluster can broadcast messages to all other members of the cluster and thus reference-broadcast techniques can be used to synchronize the cluster internally. Some nodes are members of several clusters and participate independently in all corresponding synchronization procedures. These nodes act as time gateways to translate time stamps from one cluster to the other;

(3) Tree construction: The most common solution of the multi-hop synchronization problem is to construct a synchronization tree with a single master at the root.