



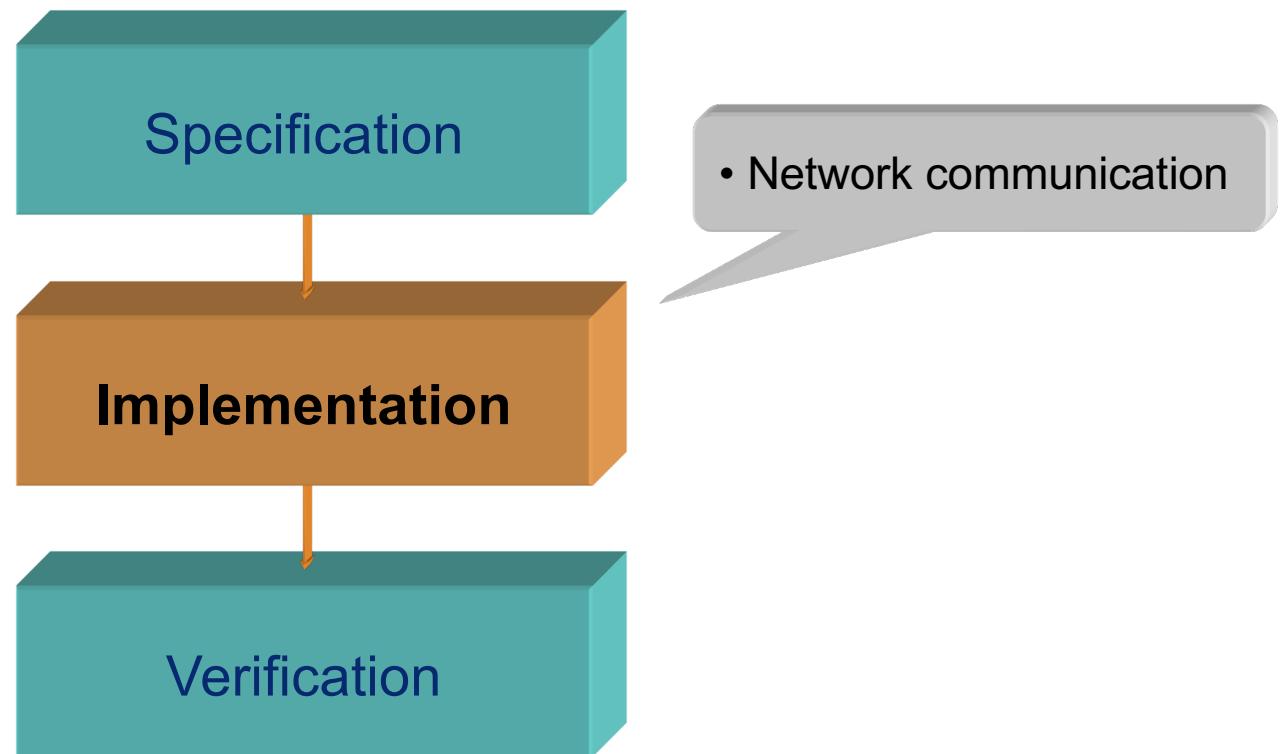
# Real-Time Systems

Lecture #8

Victor Wallsten

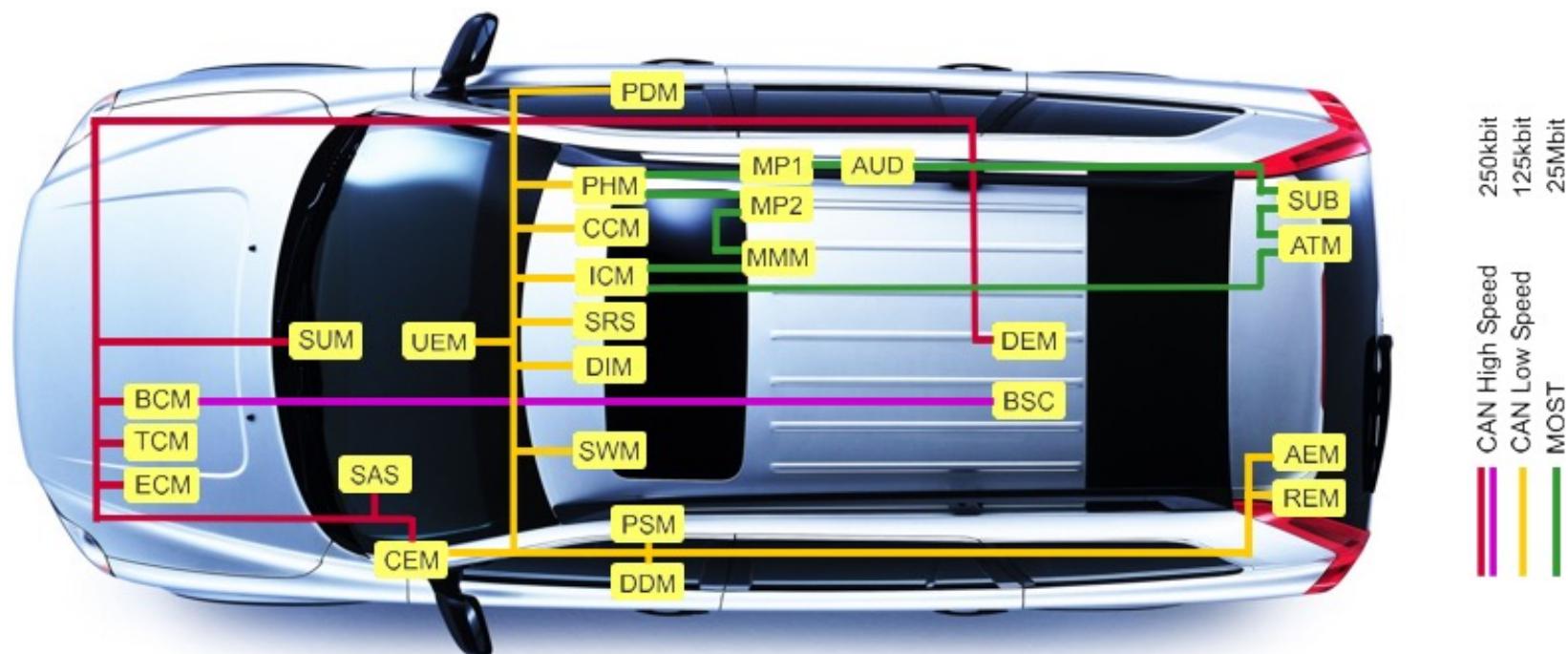
Department of Computer Science and Engineering  
Chalmers University of Technology

# Real-Time Systems

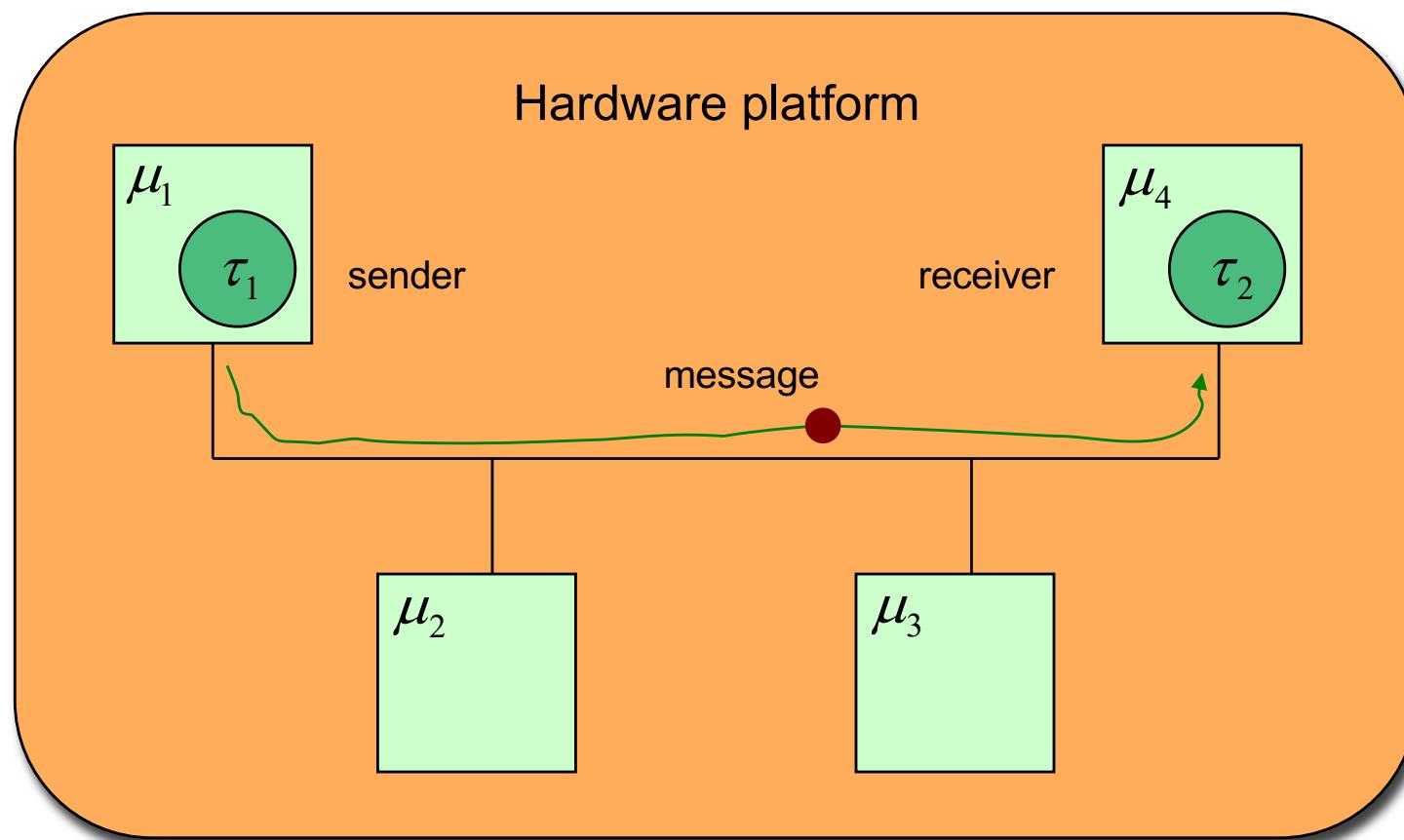


# Network communication

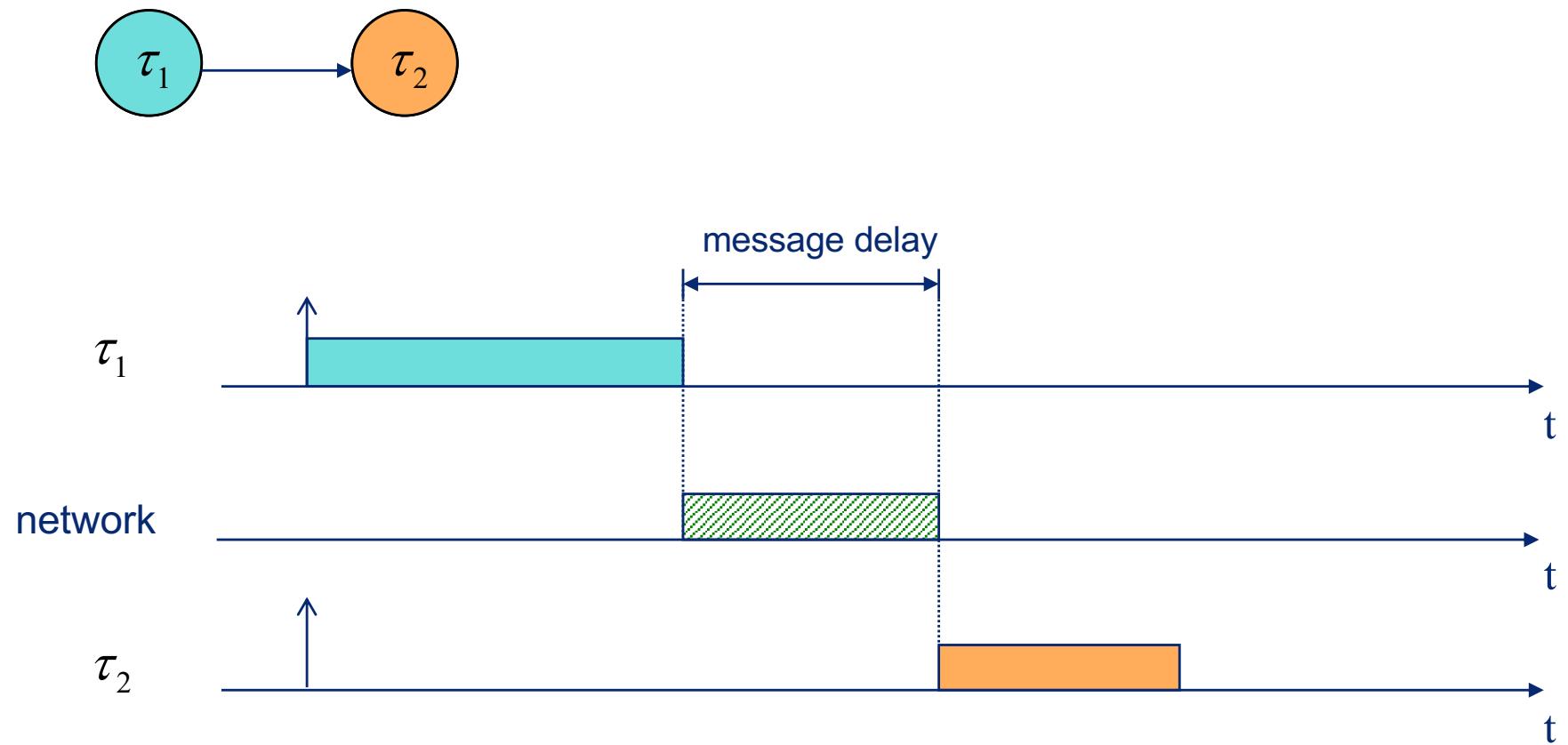
Embedded systems in the aircraft and automotive domain require support for real-time network communication



# Network communication



# Network communication



# Network communication

## Message delay:

- Message delays are caused by the following overheads:
  - Formatting (packetizing) the message
  - Queuing the message, while waiting for access to medium
  - Transmitting the message on the medium
  - Notifying the receiver of message arrival
  - Deformatting (depacketizing) the message

Formatting/deformatting overheads are typically included in the execution time of the sending/receiving task.

# Network communication

## Queuing delay:

- The cause of the queuing delay for a message depends on the actual network used. For example:
  - Waiting for a corresponding time slot (e.g., FlexRay)
  - Waiting for a transmission token (e.g., Token Ring)
  - Waiting for a contention-free transmission (e.g., Ethernet)
  - Waiting for network priority negotiation (e.g., CAN)
  - Waiting for removal from priority queue (e.g., Switched Ethernet)

To be used in a real-time system with hard timing constraints the queuing delay must be bounded.

# Network communication

## Transmission delay:

- The delay for transmitting the message is the sum of:

a frame delay

- message length (bits)
- data rate (bits/s)

$$t_{\text{frame}} = \frac{N_{\text{frame}}}{R}$$

and a propagation delay

- communication distance (m)
- signal propagation velocity (m/s)

$$t_{\text{prop}} = \frac{L}{v}$$

# Network communication

How is the message transfer synchronized between communicating tasks?

- Asynchronous communication:
  - Sending and reception of messages are performed as independent operations at run-time.
- Synchronous communication:
  - Sending and receiving tasks synchronize their network medium access at run-time.

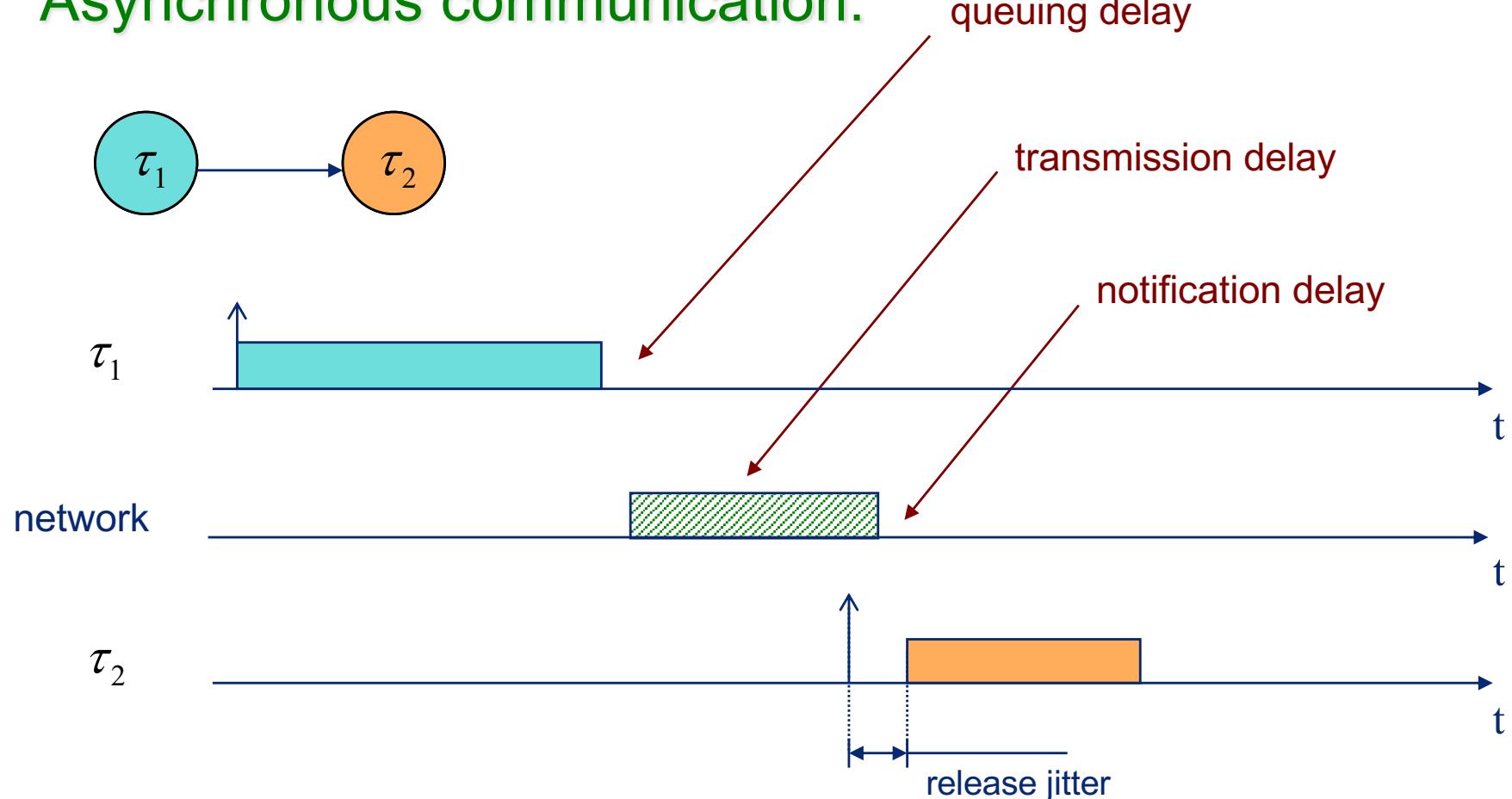
# Network communication

## Asynchronous communication

- Implementation:
  - Network controller chip administers message transmission and reception (example: CAN, Ethernet)
  - Interrupt handler notifies the receiver
- Release jitter:
  - Queuing delays at sender and notification delay at receiver cause variations in message arrival time
  - Arrival-time variations gives rise to release jitter at receiving task (which may negatively affect schedulability)
  - Release jitter is minimized by adding offsets to receiving tasks

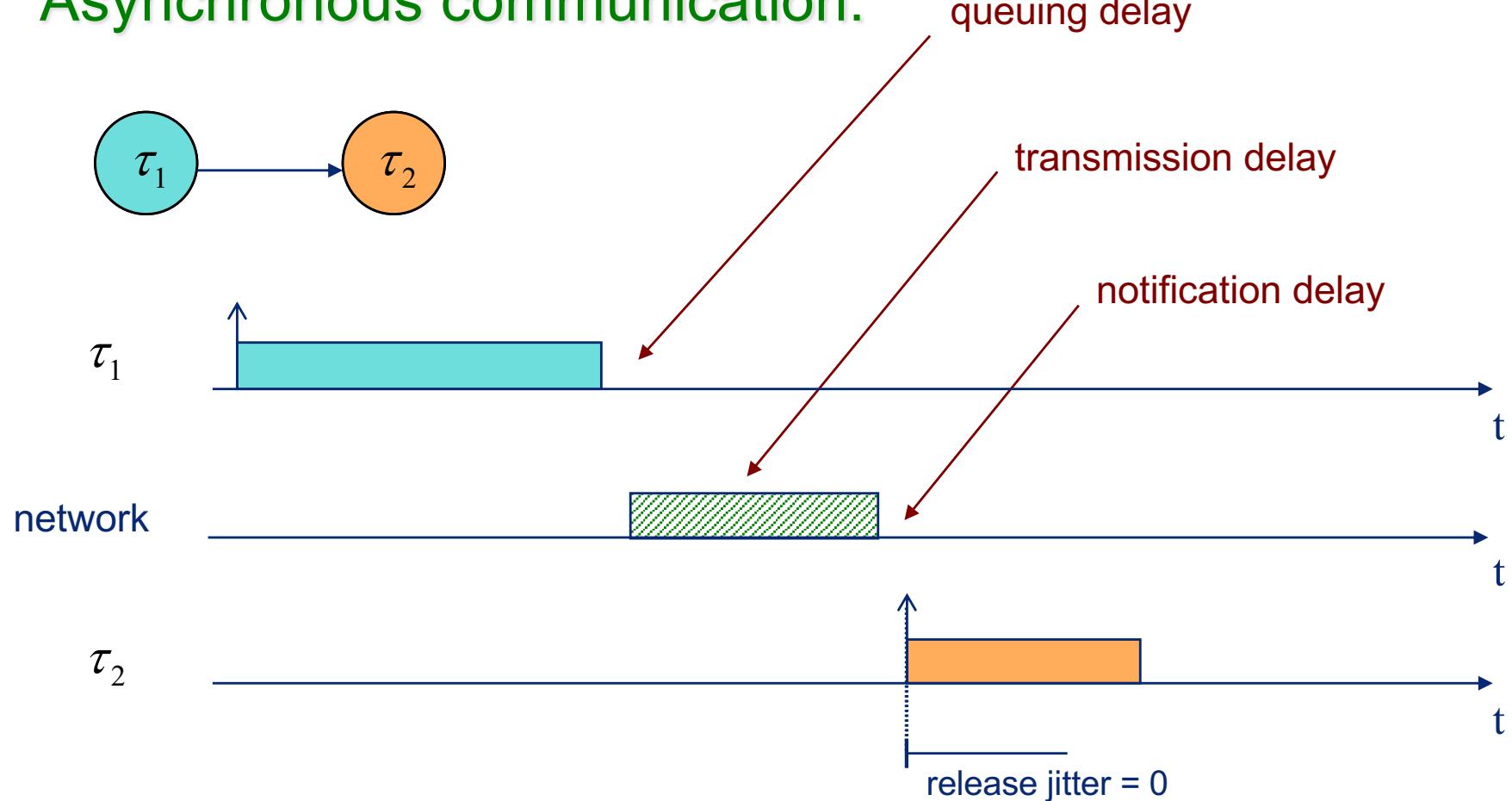
# Network communication

Asynchronous communication:



# Network communication

Asynchronous communication:



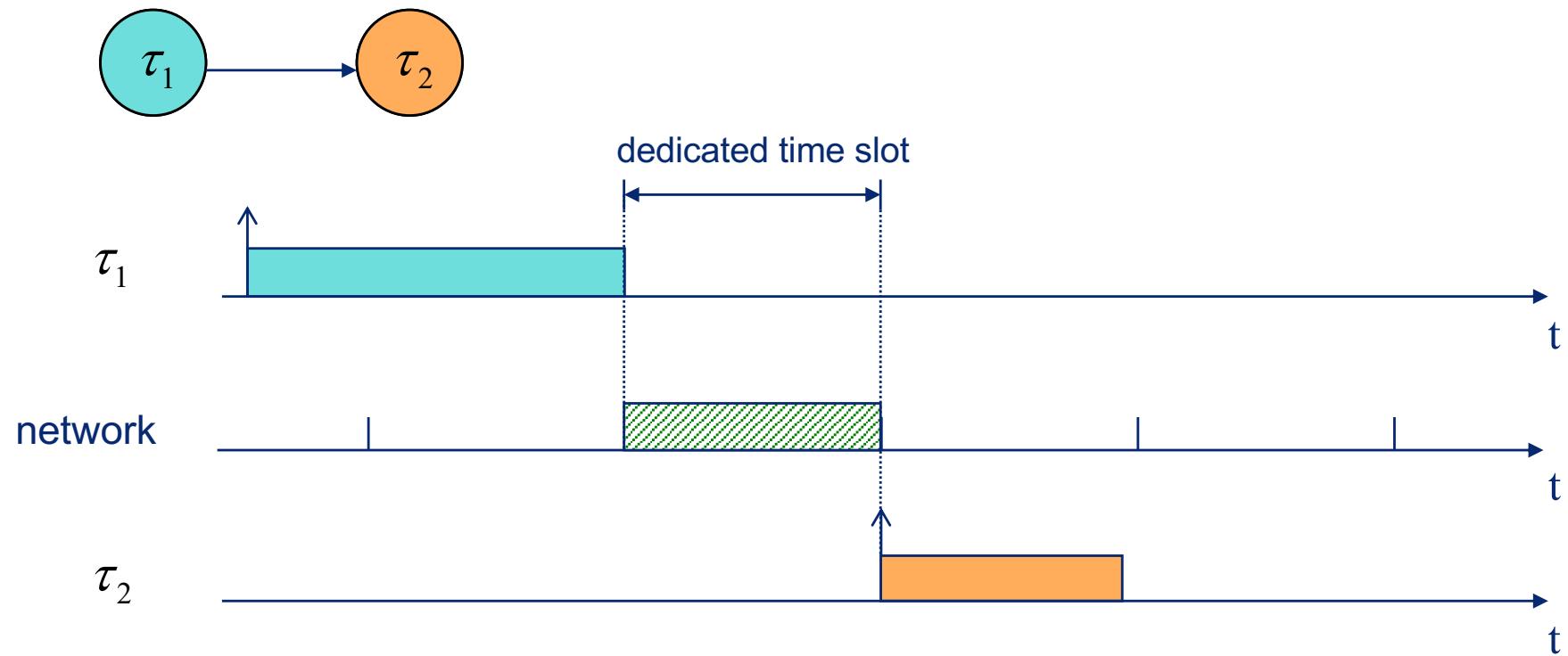
# Network communication

## Synchronous communication

- Implementation:
  - Network controller chip makes sure message transmission and reception occurs within a dedicated time slot in a TDMA bus network (example: FlexRay)
  - Off-line static (time-table) scheduling is used for matching the time slot with the execution of sending and receiving tasks
  - Queuing and notification delays can be kept to a minimum by instructing the off-line scheduling algorithm to use jitter minimization as the scheduling objective

# Network communication

Synchronous communication:



# Network communication

How is the message transferred onto the medium?

- Contention-free communication:
  - Senders need not contend for medium access at run-time
  - Examples: TTCAN, FlexRay, Switched Ethernet
- Token-based communication:
  - Each sender using the medium gets one chance to send its messages, based on a predetermined order
  - Examples: Token Ring, FDDI
- Collision-based communication:
  - Senders may have to contend for the medium at run-time
  - Examples: Ethernet, CAN

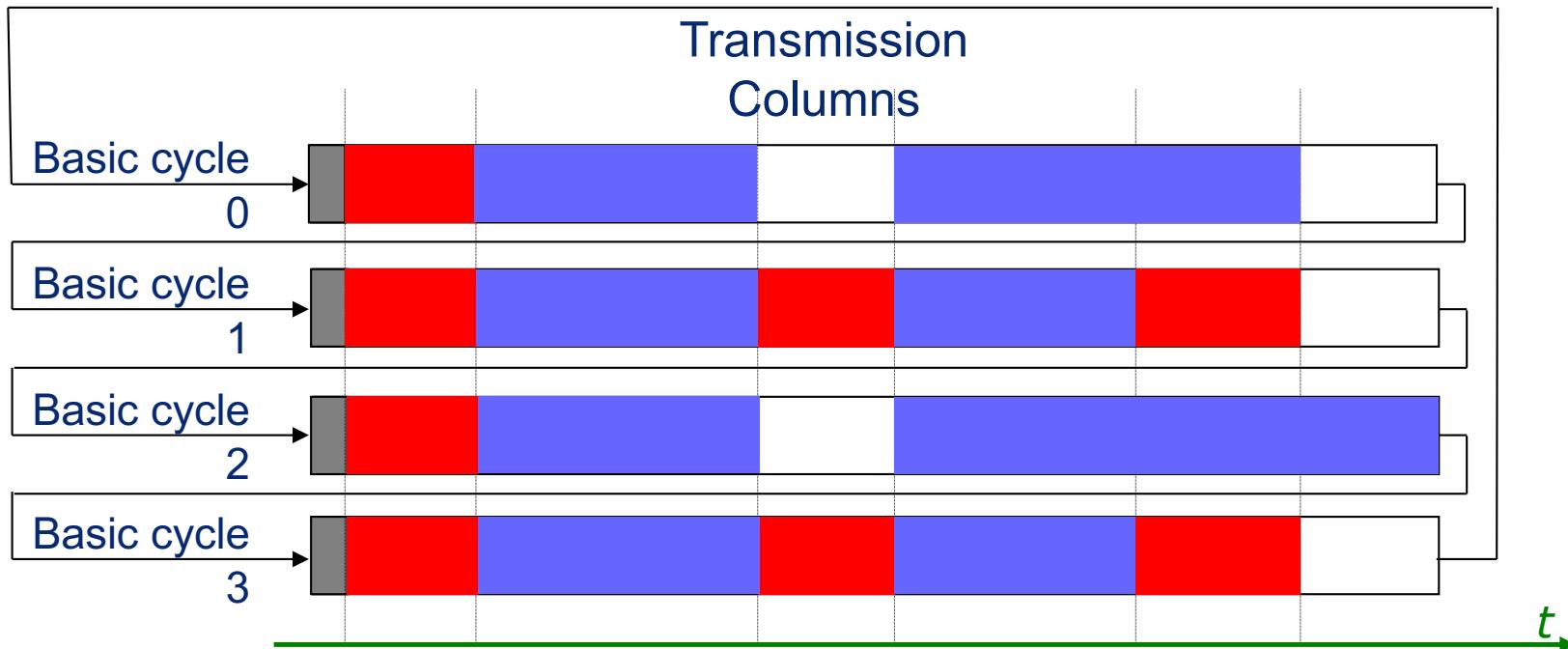
# Network communication

## Contention-free communication:

- One or more dedicated time slots for each task/processor
  - Shared communication bus
  - Medium access is divided into communication cycles (normally related to least-common-multiple of task periods)
  - Dedicated time slots provide bounded queuing delays
  - TTCAN ("exclusive mode"), FlexRay ("static segment")
- One sender only for each communication line
  - Point-to-point communication networks with link switches
  - Output and input buffers with deterministic queuing policies in switches provide bounded queuing delays
  - Switched Ethernet

# The TTCAN protocol

- "Exclusive" – guaranteed service
  - "Arbitration" – guaranteed service (high ID), best effort (low ID)
  - "Reserved" – for future expansion...



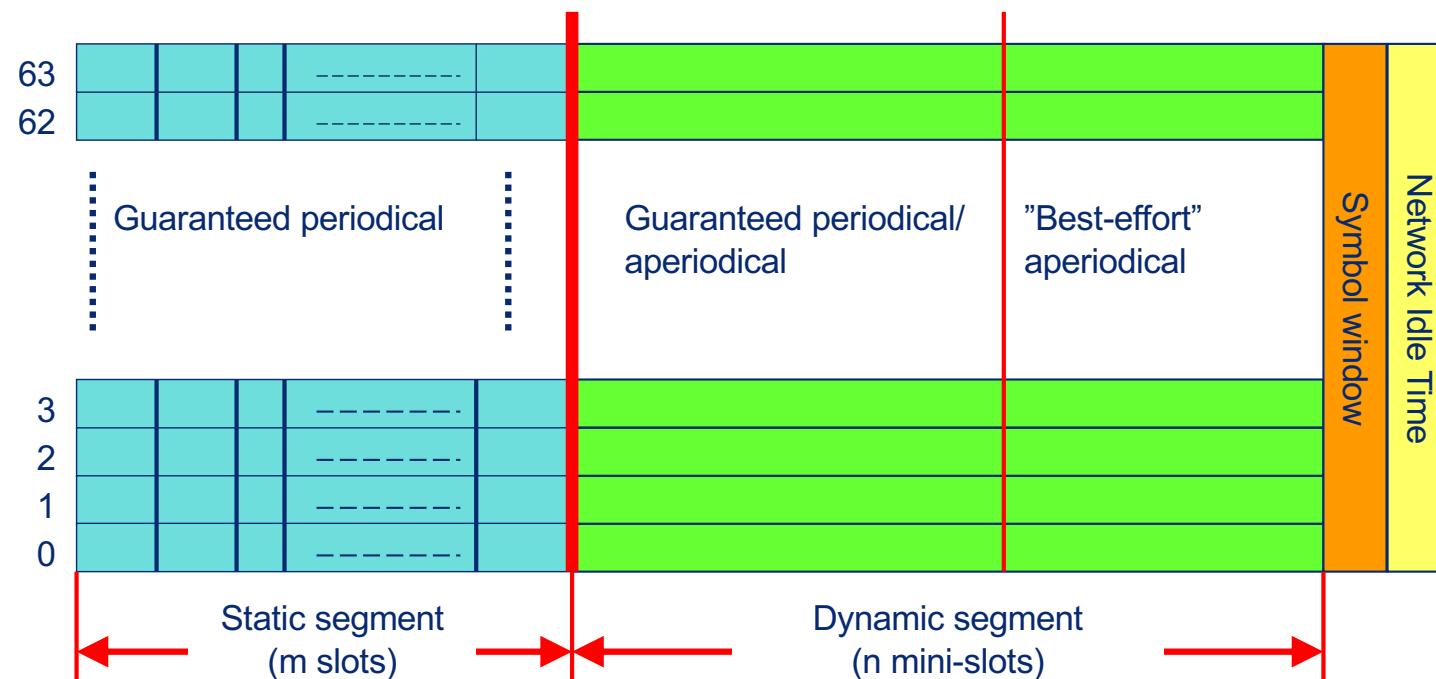
# The FlexRay protocol

■ "Static segment" (compare w/ TTCAN "Exclusive")

- guaranteed service

■ "Dynamic segment" (compare w/ TTCAN "Arbitration")

- guaranteed service (high ID), "best effort" (low ID)



Max 64 nodes on a Flexray network.

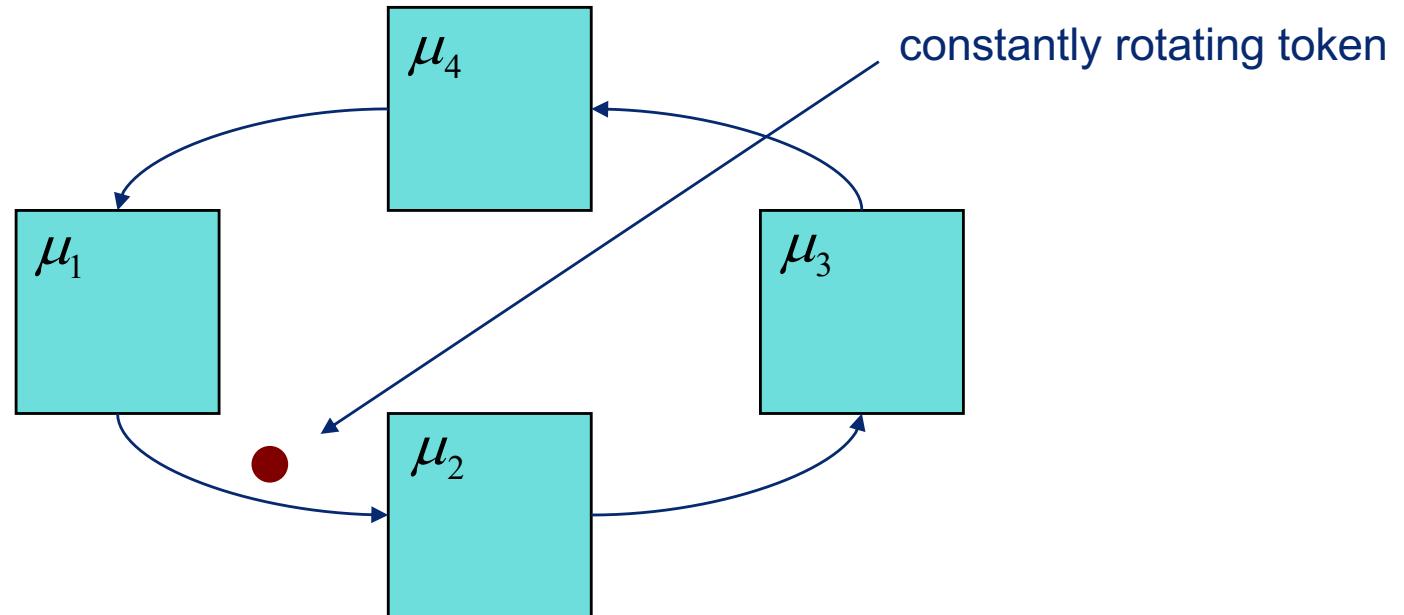
# Network communication

## Token-based communication:

- Utilize a token for the arbitration of message transmissions on a shared medium
  - The sender is only allowed to transmit its messages when it possesses the token
  - Message priorities can provide bounded queuing delays
- Examples:
  - Token Ring ([IEEE 802.5](#))
  - FDDI ([ANSI X3T9.5](#))

# Token-based communication

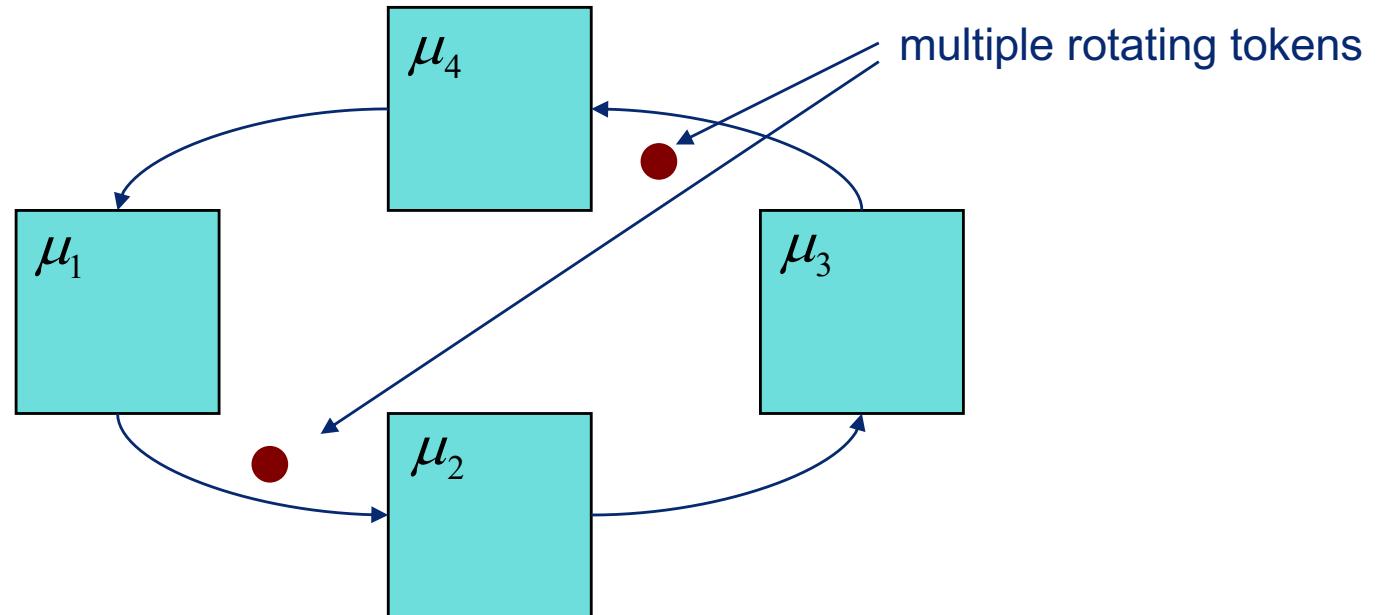
Token Ring: (IEEE 802.5)



Single rotating ring, twisted pair, 4 Mbit/s

# Token-based communication

Fiber Distributed Data Interface: (ANSI X3T9.5)



Dual counter-rotating rings, optical fibre, 100 Mbit/s

# Network communication

## Collision-based communication:

- Utilize collision-detect mechanism to determine validity of message transmissions on a shared medium
  - The sender tries to send messages independently of other senders' intention to do so
  - Attempts may be done at any time or when some specific network state occurs
- Examples:
  - Ethernet w/ multiple senders (**IEEE 802.3**)
  - CAN (**ISO 11898**)

# Collision-based communication

Ethernet protocols w/ multiple senders:

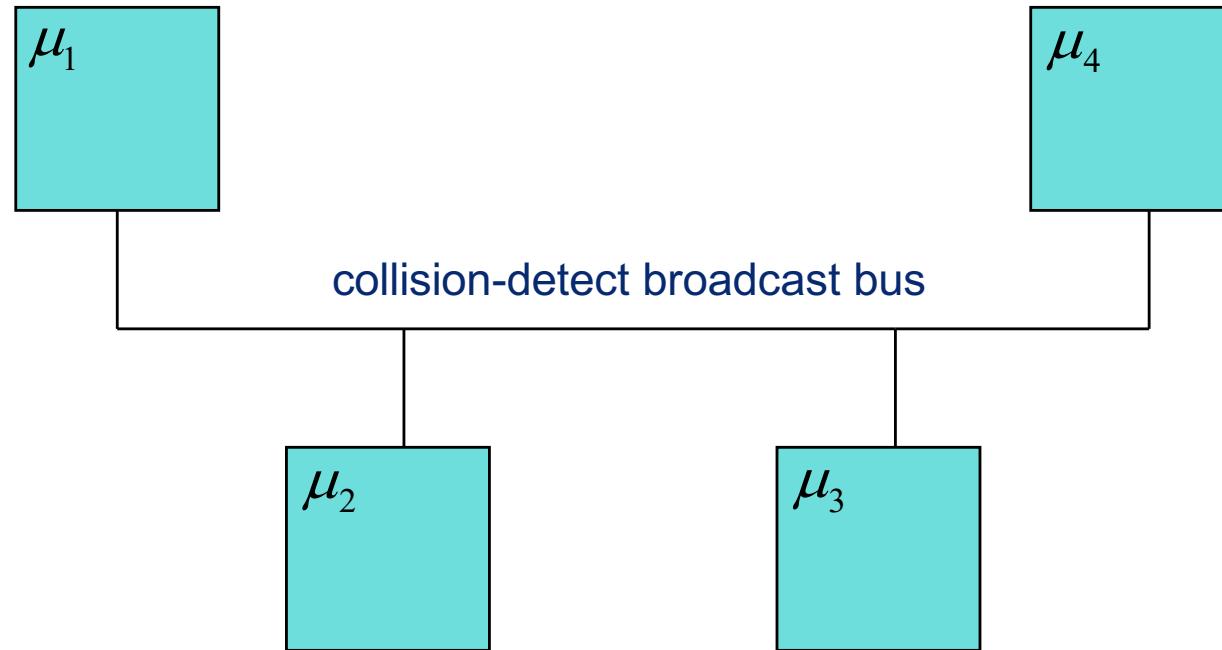
- Senders attempt to send a complete message
- If messages collide, all transmissions are aborted
- After collision, re-transmission is made after a random delay

Message queuing delay can in general not be bounded!

Therefore, these protocols do not give any guarantees for meeting imposed message deadlines!

# Collision-based communication

Controller Area Network (CAN): (ISO 11898)



Broadcast serial bus, dual wire (resistor terminated), 1 Mbit/s

# Collision-based communication

## Controller Area Network (CAN):

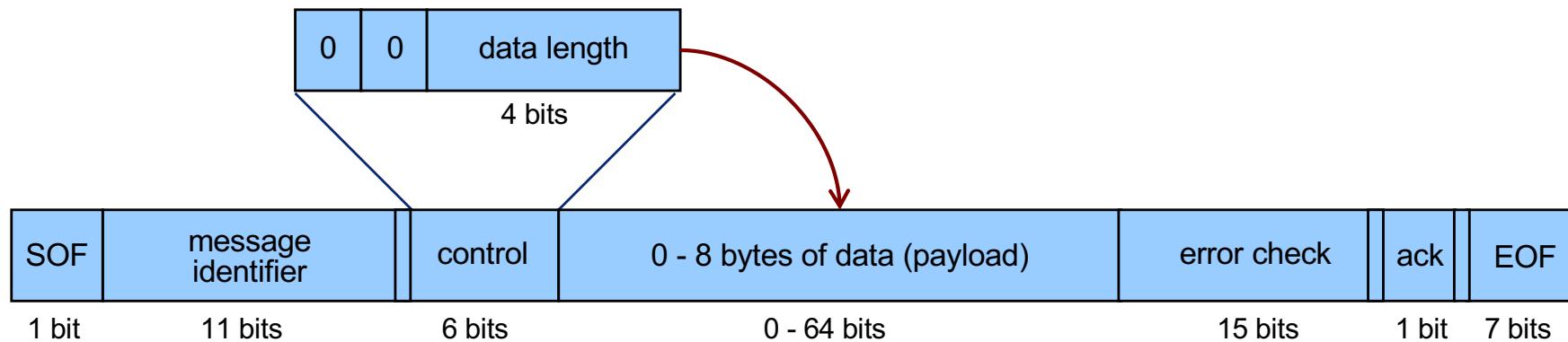
- Senders transmit a message header (with an identifier)
- If messages collide, a hardware-supported protocol is used to determine what sender will be allowed to send the rest of the message; transmissions by other senders are aborted

Message queuing delay can be bounded with appropriate identifier assignment.

Therefore, this protocol makes it possible to meet imposed message deadlines.

# The CAN protocol

CAN message frame format: (short format)

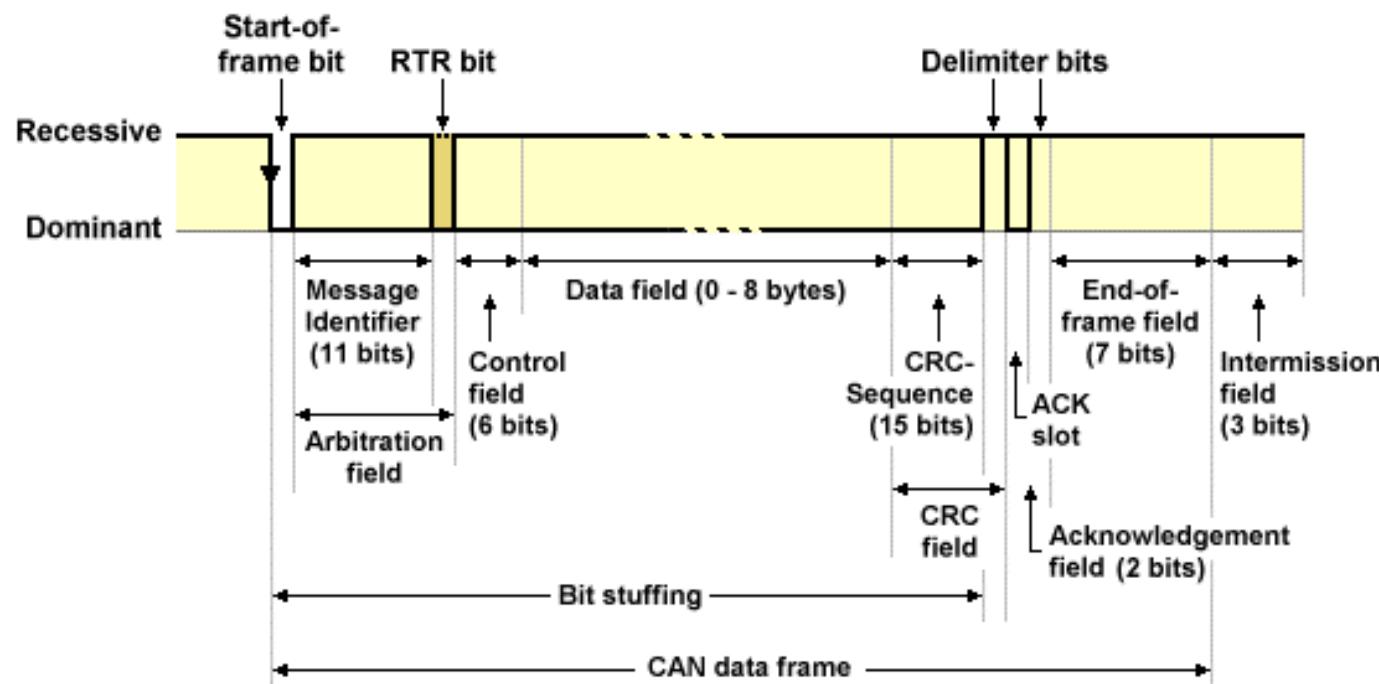


Message identifier can be used for several purposes:

- enable receiver to filter messages (original purpose)
- assign a priority to the message (low number  $\Rightarrow$  high priority)

# The CAN protocol

CAN message frame format: (short format)



# The CAN protocol

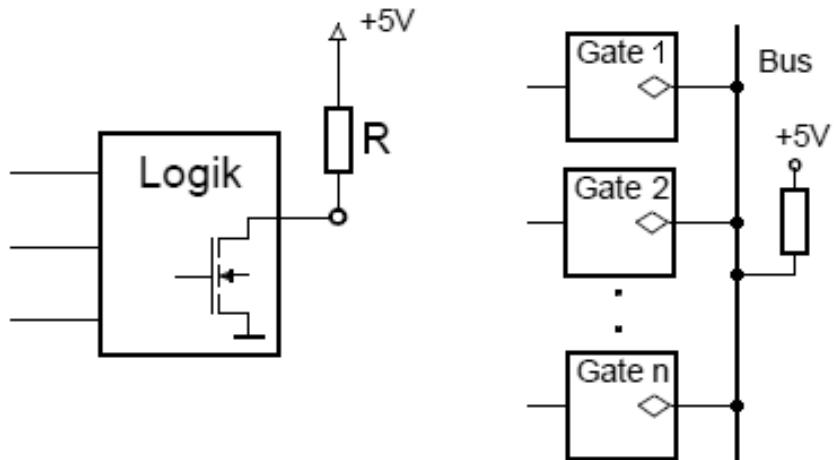
## CAN protocol: (binary countdown)

### Wired-AND:

Each node monitors the bus while transmitting.

If multiple nodes are transmitting simultaneously and one node transmits a '0', then all nodes will see a '0'.

If all nodes transmit a '1', then all nodes will see a '1'.



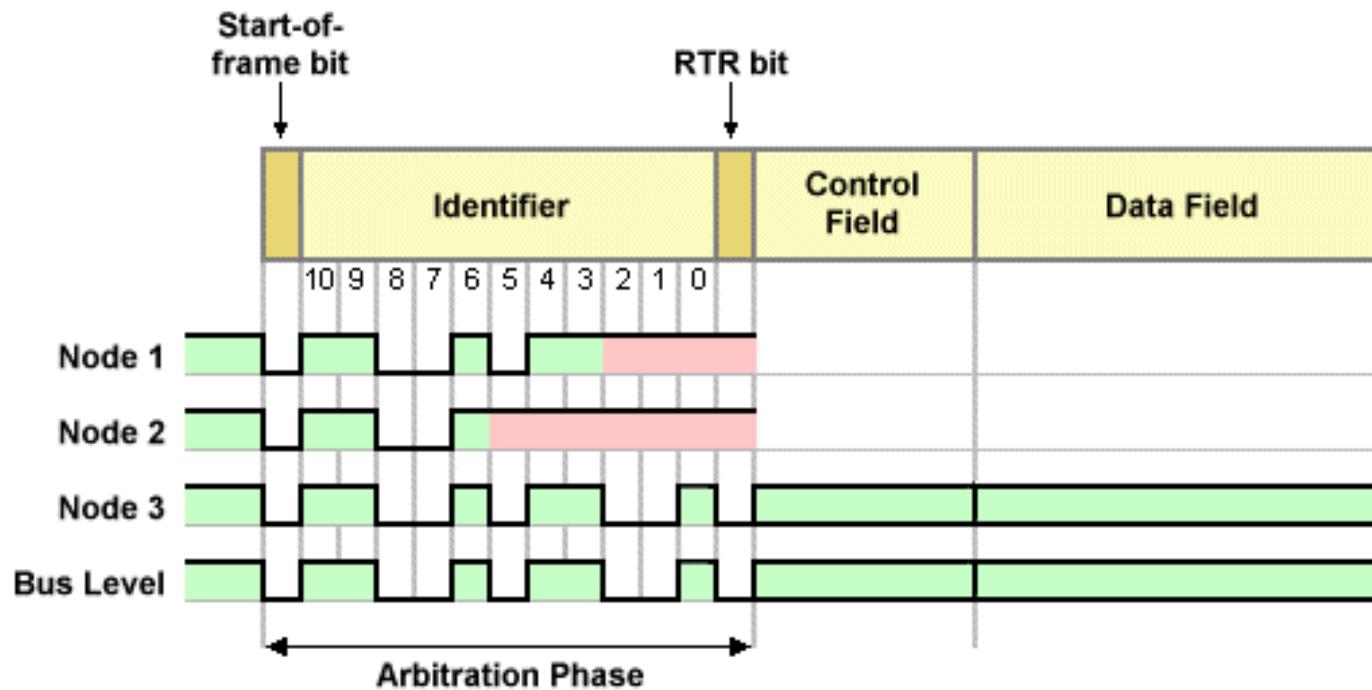
# The CAN protocol

## CAN protocol: (binary countdown)

1. Each node with a pending message waits until bus is idle.
2. The node begins transmitting the highest-priority message pending on the node. Identifier is transmitted first, in the order of most-significant bit to least-significant bit.
3. If a node transmits a recessive bit ('1') but sees a dominant bit ('0') on the bus, then it stops transmitting since it is not transmitting the highest-priority message in the system.
4. The node that transmits the last bit of its identifier without detecting a bus inconsistency has the highest priority and can start transmitting the rest of the message frame.

# The CAN protocol

CAN protocol: (binary countdown)



# Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler:

1. Define class `Can`, and add state variables for:
  - the hardware base address of the device
  - call-back information for a method if data received by the handler needs to be taken care of by the user-level code (the call back should be done using an `ASYNC()` call)
  - necessary local storage (buffers, queues, etc)

2. Define a symbol `CAN_PORT0` representing the hardware base address of the device.

```
#define CAN_PORT0 device.hardware_address
```

3. Create an object `can0` of class `Can`, and initialize it with:
  - the hardware base address `CAN_PORT0`
  - any possible call-back information

# Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

In file 'application.c':

```
App app = { initObject(), 0, 'X' };

void receiver(App*, int);

Can can0 = initCan(CAN_PORT0, &app, receiver);

void receiver(App *self, int unused) { // call-back function
    CANMsg msg;
    CAN_RECEIVE(&can0, &msg);
    SCI_WRITE(&sci0, "Can msg received: ");
    SCI_WRITE(&sci0, msg.buf);
}
```

# Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

4. Write an interrupt handler as a method `can_interrupt` and associate it with the object.
5. Declare a symbol `CAN_IRQ0` and assign to it the TinyTimber kernel's logical number of the hardware interrupt:

```
#define CAN_IRQ0 interrupt_logical_number
```

6. Inform the TinyTimber kernel that the method is a handler for interrupt `CAN_IRQ0`, by making a call to

```
INSTALL(&can0, can_interrupt, CAN_IRQ0);
```

This should be done before the call to `TINYTIMBER()`

# Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

7. Provide an operation `CAN_INIT()` that takes care of performing any remaining initialization of the device.
8. Call `CAN_INIT()` in the “kick-off” method that was supplied as argument to the `TINYTIMBER()` call.

# Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

In file 'application.c':

```
void startApp(App *self, int arg) {
    CANMsg msg;
    CAN_INIT(&can0);
    ...
    CAN_SEND(&can0, &msg);
}

int main() {
    INSTALL(&can0, can_interrupt, CAN_IRQ0);
    TINYTIMBER(&app, startApp, 0);
}
```

# Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

In file 'canTinyTimber.h':

```
typedef unsigned char uchar;

typedef struct {
    uchar msgId; // Valid values: 0-127
    uchar nodeId; // Valid values: 0-15
    uchar length;
    uchar buff[8];
} CANMsg;
```

CANMsg holds the user-relevant parts of the CAN message frame.

- nodeId holds the four least-significant bits of the identifier field
- msgID holds the seven most-significant bits of the identifier field