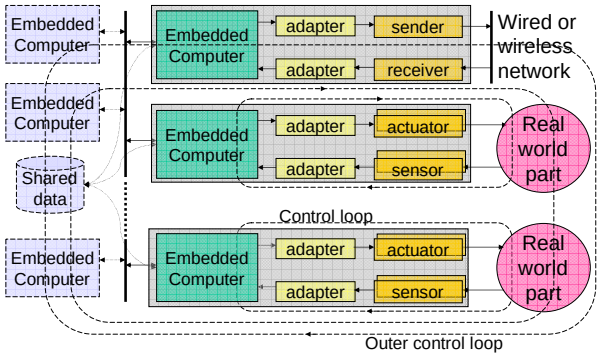
Distributed systems

- A distributed system (DS) is a set of physically separated computers connected via a communication network
- Some computers and other resources are unique and other exist in more than one copy
- A natural reason for DS is that the system parts are spread apart in the real physical world
- Other reasons are high requirements on e.g., computing power, dependability and scalability
- Distributed (local/decentralized) processing can limit the data communication needs and costs for wiring between these parts

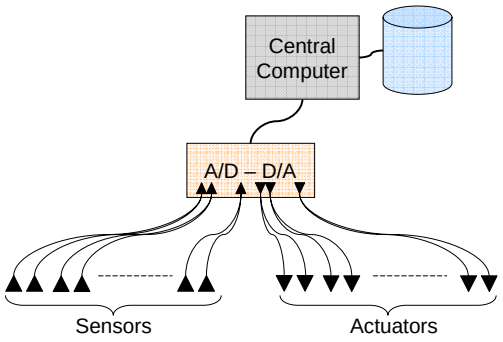
Example – distributed control system



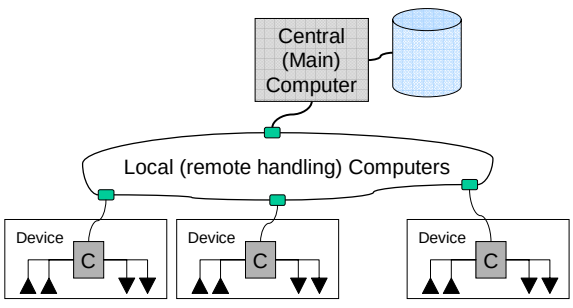
Distributed resources

- The resources of a distributed system must be able to communicate via wired or wireless means
- Network distributed resources are:
  - Processors
  - Memory (place to store data)
  - Input/Output devices
  - Sensors
  - Actuators
  - Data (Task state, measurements, etc.)
- Often the processor and specific communication hardware activates and drives the communication

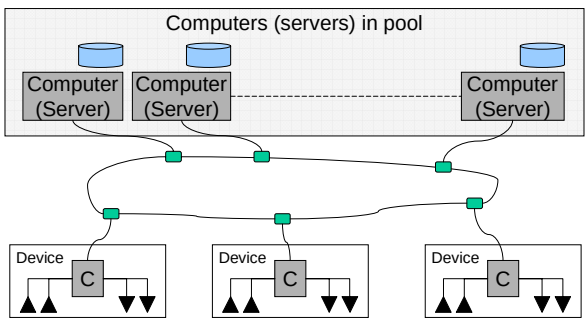
Sensors & actuators distributed



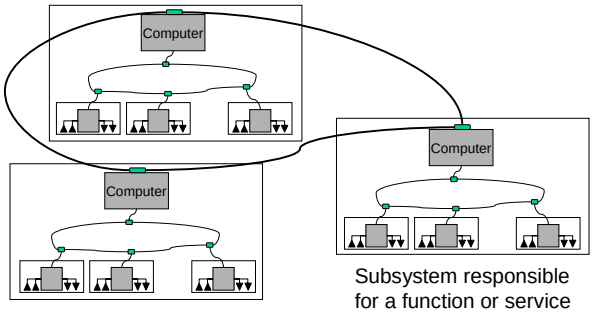
Locally handled sensors & actuators



Distributed control and server pool



Function partitioned control



Reliable Computer Systems

System reliability can be improved by:

- Minimizing HW and SW design errors
- Minimizing HW manufacturing errors
- Increasing HW lifetime
- Redundant HW and SW
- Data replication
- Fault recovery

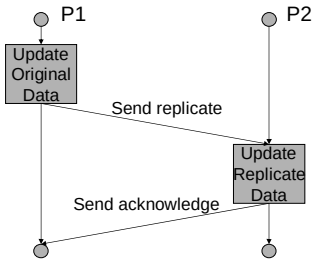
Data replication

- Important data must be kept in a safe store
- This means that it must be stored in more than one copy
- To protect from the consequences of hardware faults the copy must be stored in a second well separated physical location
- Protection against destruction (e.g. due to fire) require physically distant placement of the secondary physical backup location

Data importance and redundancy

- Individual measurement samples can be acquired or sent again if lost
- A counter of acknowledgment signals is more sensitive
- We must provide enough redundancy for such important state data
- Safety can be improved by use of redundant (replicating) data formats
- Data can be backed up by both by source (server) and by client(s) to provide redundancy

Replication takes time and capacity



Replication means:

- communication resources are used twice
- processor and scheduler are disturbed twice

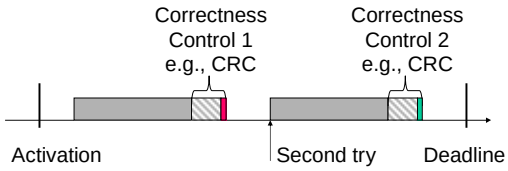
Delayed mass replication

- We can save capacity by replication of several original data items at a time
- The replication is then delayed for some time until a package (or bulk) of data is produced
- During this delay period we have reduced redundancy and thus tolerance to faults
- But this still adds redundancy if the delay is just a fraction of the total lifetime of the data
- To replicate a bulk of data items can be more efficient than to replicate individual data at once

Replication

- Assume that the life time of data is much longer than the time needed to produce an item and a corresponding replicate (copy = P + R
- Time for production and replication of N items send to a safe store in bulk = N * (P + R_B) where R_B < R (per item)
- Mean time between SW program faults usually much longer than data life time
- Mean time between HW faults much longer than mean time between SW faults
- **Under above circumstances it is better to replicate in bulk than not to replicate at all**

Replication by retry



- If a task is very important one can make more than one attempt to perform a task correctly
- This applies both to computation and communication tasks

Remote control issues

- Physical distance and limited bandwidth between parts lead to signal propagation delays and synchronisation problems
- Tasks executing on the same and on different processors interfere with each other either directly if communicating or when competing for shared resources
- And this in ways that are difficult to predict

Transmission delay

N = message or packet length (bits)

B = bandwidth (bits/s)

L = communication distance (m)

V = signal propagation velocity (m/s)

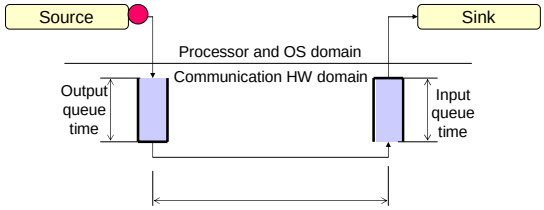
Transmission delay = N/B + L/V

Queuing delay

Depends on:

- Bandwidth and message length
- Priority
- Load
- Variation of load (probability distribution function)
- Selection principle (e.g. round robin or weighted fair queuing)
- Other tasks with same or higher priority or lower that blocks a resource

Cost of inter task communication



Transport time = message length (packet) / bandwidth +
communication distance / signal velocity* +
queue time (output and input) +
task scheduling +
dispatch time

(*signal travels with speed less than the speed of light)

Message to send selection

- first come first served (round robin)
- last come first served
- removal from priority queue using (EDD-D) *Earliest Due Date for Delay*
- weighted fair queuing (if deadline information is not available)

Media access

- allocated time slot (TDMA)
- transmission token (Token ring)
- contention avoiding principles such as CS/CA/CD (Ethernet)
- shared CS/CA/CD message broadcast with identity priority (CAN)

Scheduling of tasks and messages

Integrated scheduling

- Scheduling of task processing and message communication are regarded as complementary jobs causing a dependency
- Produce -> communicate -> consume
- Requires compatible dispatching strategies

Separated scheduling

- Scheduling of task processing and message communication are regarded separately
- Requires tolerance to more of queuing (buffer) delay
- Allows for different dispatching strategies

Scheduling of message (as job)

- Treat communication channel as resource
- Treat sending (communication) of message as a new job (single invocation of a task instance)
- Treat reception of message as (triggering of) a new job (single invocation of a task instance)

Real-time communication protocols

Time division multiple access (TDMA):

- Processors or tasks have fixed time slots
- Need not contend for medium access at run-time

Self organized TDMA:

- All or some time slots allocated dynamically on demand
- Reuse of not used time slots (by other tasks)
- Organized contention for time slots

Token-based access:

- Processors or tasks using the medium gets one chance per token to send its messages, based on a predetermined order
- Example: FDDI, Token Ring (IEEE 802.5)

Carrier sense, collision avoidance, collision detection multiple access:

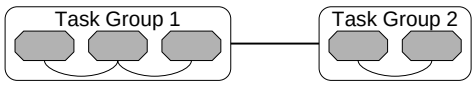
- Messages may collide with some probability
- Processors may have to contend for the medium at run-time
- Example: Ethernet (IEEE 802.3), Controller Area Network (CAN)

Task partitioning

- A large task can be divided into several smaller subtasks to be executed on a multi-processor (shared memory) computer or a distributed (parallel) computer
- This can be done mechanically by use of a computer program guided by a selected policy for task partitioning
- The result can be optimized with respect to the selected policy for task partitioning

Grouping of tasks (philosophy)

- Tasks may depend on each other directly due to sharing of common resources (e.g. data) and indirectly due to the cost for communication and the cost for task interruption (preemption)
- **Dependent tasks should be kept together** in the same group and more independent tasks be kept apart in different task groups (clusters)

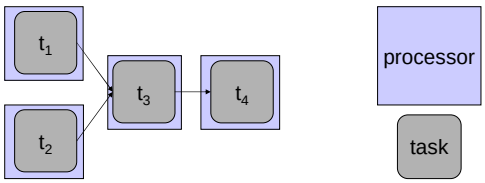


Grouping of processors

- A set of processors with “similar” properties can be provided to a set of tasks as a **pool of processing resources**
- This requires that all tasks can select any from a set of processors kept in a pool and use it as a computing resource for the given purpose
- **General processors** have no specific HW or SW resources and are thus mainly computational resources
- **Dedicated processors** are connected to specific (unique) HW or SW resources or physical locations
- We have general and dedicated processors as well as general and dedicated pools of processors

Task distribution

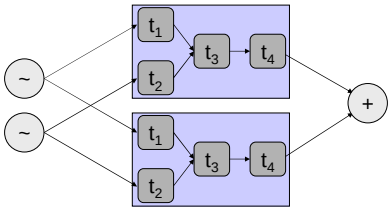
Function separation



The overall function depends on all 4 processors and all 4 tasks to work (series connected)

Task Distribution, cont.

Load sharing What is the resulting fault probability?

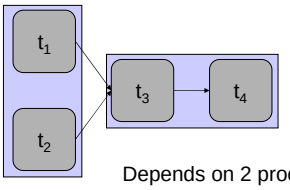


Redundant Function with 50% capacity remaining if only one processor is faulty

What is the probability (for a total stop)?
(Think in terms of parallel and series connected probabilities)

Task Distribution, cont.

Partly function separated and partly load shared



Depends on 2 processors a 4 tasks

What is the resulting fault probability?

Fault probability

- Probability for an outcome X is written P(X)
- Relations between independent outcomes can be illustrated with Wenn diagrams
- Typical outcomes that we talk about are:
  - X is faulty
  - X works
  - X has succeeded
  - X arrives

Fault probability

The probability that the function of A and B both are available (required for the system to work) can be expressed:

$$P(A\,\&\,B) = P(A) \cap P(B)$$

Provided that the function of A does not depend on the function of B we can express this as:

$$P(A) \cap P(B) = P(A)P(B)$$

If A and B are redundant (parallel) the probability that at least one of A and B works is:

$$P(A) \cup P(B) = P(A) + P(B) - P(A) \cap P(B)$$

Fault probability

For a system with three tasks or components A, B and C we get the probability that at least one works:

$$P(A) \cup P(B) \cup P(C) =$$

$$P(A) + P(B) + P(C)$$
$$- P(A) \cap P(B) - P(A) \cap P(C) - P(B) \cap P(C)$$
$$+ P(A) \cap P(B) \cap P(C)$$

Fault probability, cont.

Some fault events are independent and some are mutually exclusive to each other

Due to deeper functional dependencies we however sometimes also get conditional (dependent) probabilities

The (conditional) probability for event A to happen but requiring event B also to happen, is:

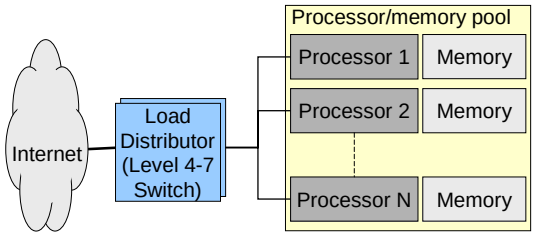
$$P(A \mid B) = \frac{P(A) \cap P(B)}{P(B)}$$

Many jobs of the same kind

- Servers used for web services and phone calls* handle many (10 - 1000nds of) similar short jobs each second served by “processors in pool”
- The processors can have access to all “web or call data” in one big shared data pool
- Alternatively each processor serves a section or sub-pool of “web or call data”

**A call in this context is a service session*

Web service computer pool



Each processor may support a pool of threads

Static allocation of tasks

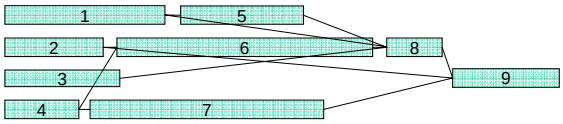
- Tasks can be statically allocated to:
  - a dedicated processor
  - a pool of processor (dynamic allocation within pool)
  - a selected processor within a pool
- Allocate the tasks over processors so that these can be utilized evenly, with respect to the processors capacity
- The total execution time requirement for a group of tasks should be proportional to the capacity of the allocated processor or pool of processors

Dynamic allocation of new jobs

- A load handler and admission controller should distribute new jobs (handled by task instances) over processors in a pool so that the processors are utilized evenly and not will become overloaded
- New arriving jobs should be allocated to the least loaded processor within a pool of processors to maintain a balanced dynamic processor utilization
- The total execution time requirements of arriving jobs should be balanced to the total capacity of the processor pool

Dependency graph

- The data dependencies between jobs needed to carry out a larger job can be illustrated in a graph
- Cyclic tightly coupled or dependent jobs with the same period can be aggregated into larger jobs that can execute on the same processor

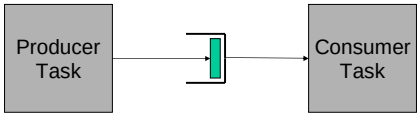


Decoupling by use of buffers

- Data depending tasks can be decoupled and made independent of each other, from a scheduling point of view, by use of buffers
- The tasks still have their own scheduling requirements and are still dependent as producers and consumers of data

Use of buffers, cont.

- Buffers (queues) reduce scheduling and communication dependencies, provided that data/messages communicated are sent within accepted periods and deadlines



Use of buffers, cont.

- This in its turn requires that the data can be delayed with the time interval that buffering and communication services introduce in worst case situations
- A worst case is a critical instant when all higher priority jobs need the communication media (and/or a shared buffer) at the same time

Summary

- If possible use buffers to reduce timing dependencies and thus simplify job partitioning and job allocation
- Gather communication dependent tasks in similarly sized groups (from a processor utilization point of view)
- Schedule each group of jobs using e.g., static RM or dynamic EDF scheduling