

Load Balancing and Resource Allocation

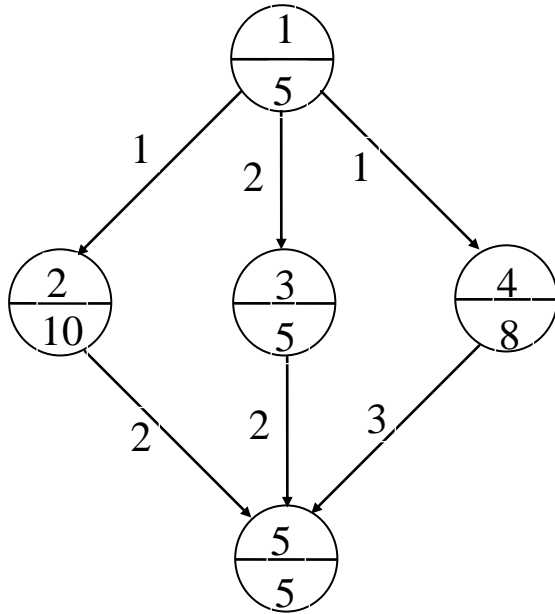
Moler's law and Sullivan's theorem give upper bounds on the speed-up that can be achieved using multiple processors. But to achieve these speed-ups we need to "efficiently" assign the different concurrent processes that make up a concurrent program on the available processors. This is called *Load Balancing*.

Load balancing is a special case of the more general *Resource Allocation Problem* in a parallel or distributed system. In the load balancing situation the resources are the processors.

Before we clarify the load balancing problem we need to formalise out models of the concurrent program and the concurrent system.

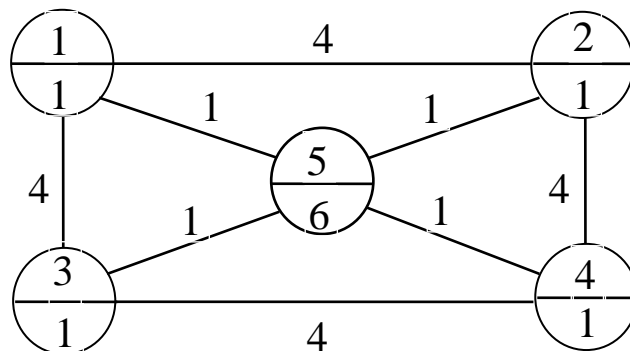
Task Graph

A *task graph* is a directed acyclic graph where the nodes represent the concurrent processes that form the concurrent program and the edges between the nodes represent the communications and synchronisation between the processes. The weight of a node is the computational load of the process the node represents, and the weight of an edge between two nodes is the amount of communications between two processes represents by the two nodes.



Processor Graph

The *processor graph* defines the configuration of the parallel or distributed system. Each node represents a processor and the weight of each node is the computation speed of the processor represented by the node. The edges between nodes represent the communication links between the processors represented by the nodes. The weight of an edge is the speed of the communications link the edge represents.



The Problem

To partition a set of interacting *tasks* among a set of interconnected *processors* in order to maximise “performance”.

Informally, the basic idea to load balancing is to attempt to balance the load on all processors in such a way as to allow progress by all tasks on all processors to proceed at the same rate.

However formally maximising “performance” can be defined as:

- minimising the makespan, C_{\max} , where the makespan is defined as the maximum completion time of any of the n tasks.

$$\min(C_{\max}) = \min(\max_{1 \leq i \leq n} C_i)$$

- minimising the response time,
- minimising the total idle time, or
- any other reasonable goal.

A general assumption that is made is that the communications between tasks on the same processor is orders of magnitude faster than communications between two tasks on different processors. Hence intra-processor communications is deemed to be instantaneous.

Allocation and Scheduling

Load Balancing has two aspects:

- the *allocation* of the tasks to processors, and
- the *scheduling* of the tasks allocated to a processor.

The current view is that allocation is more important than scheduling, particularly with some common processor configurations. Therefore some load balancing algorithms only address allocation.

Complexity of the problem

The problem of determining if there exists an allocation of n arbitrarily intercommunicating tasks, constrained by precedence relationships, to an arbitrarily interconnected network of m processing nodes, which meets a given deadline is an NP-complete problem. The problem of minimising the makespan of a set of tasks, where any task can execute on any processing node and is allowed to pre-empt another task, is NP-complete even when the number of processing nodes is limited to two.

A taxonomy of load balancing algorithms

The following hierarchical taxonomy of load balancing algorithms is due to Casavant and Kuhl.

Not all characteristics of load balancing algorithms fit nicely into a hierarchy. Some of the non-hierarchical classifications are:

One-Time Assignment vs. Dynamic Reassignment

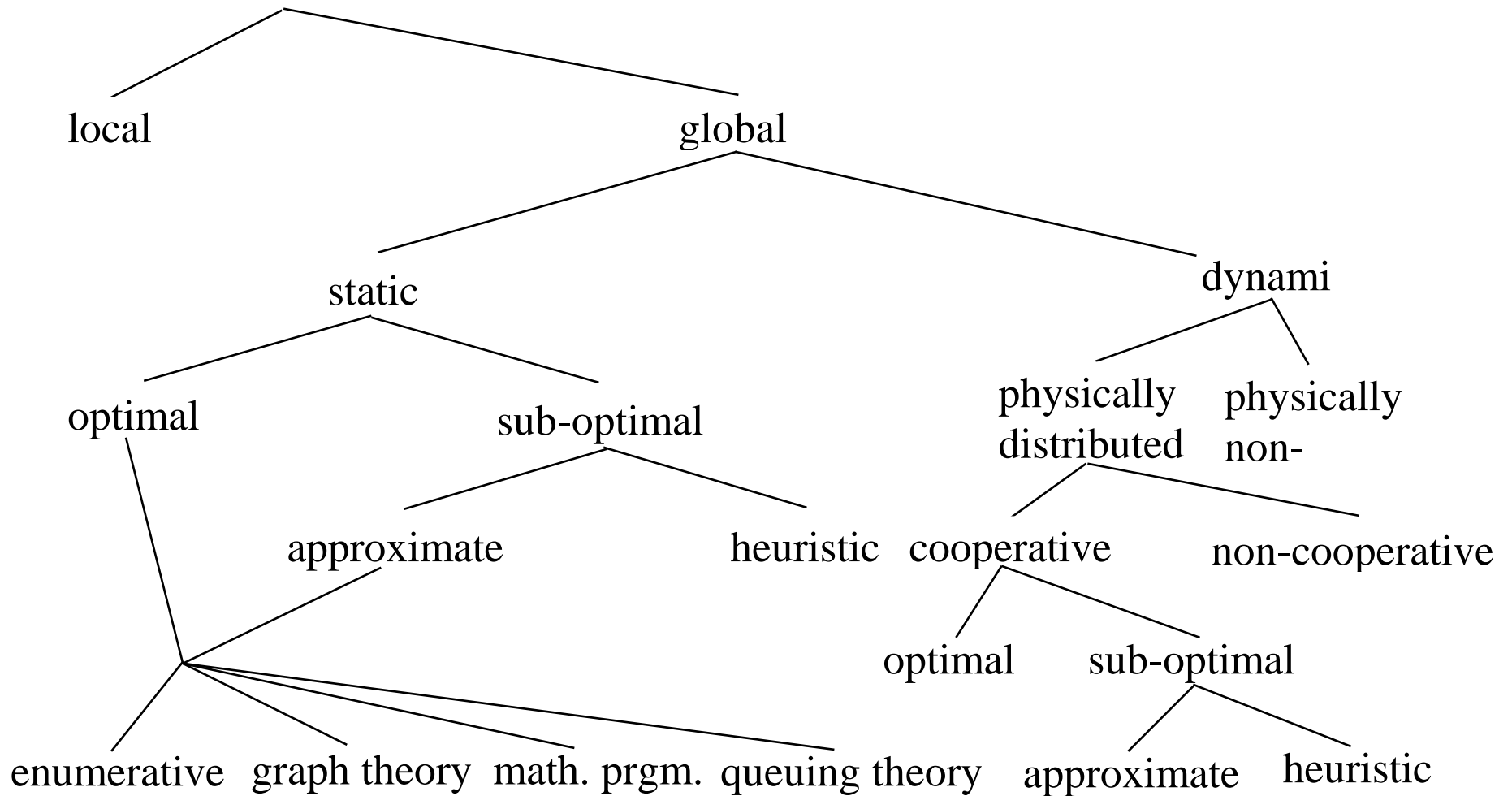
Is the assignment fixed once it is made (static), or can it use runtime state information to make load-balancing decisions (dynamic) (i.e. can tasks be moved from one processor to another as the system state changes?)

Dynamic Algorithms \neq Adaptive algorithms

There are two kinds on dynamic algorithms: Adaptive and Non-Adaptive:

Adaptive and Non-adaptive algorithms

Adaptive algorithms is one in which the algorithm parameters change according to current and previous system behaviour. Non-Adaptive algorithms can only use information about the run-time state.



Casavant & Kuhl's Taxonomy

Coffman's Algorithm

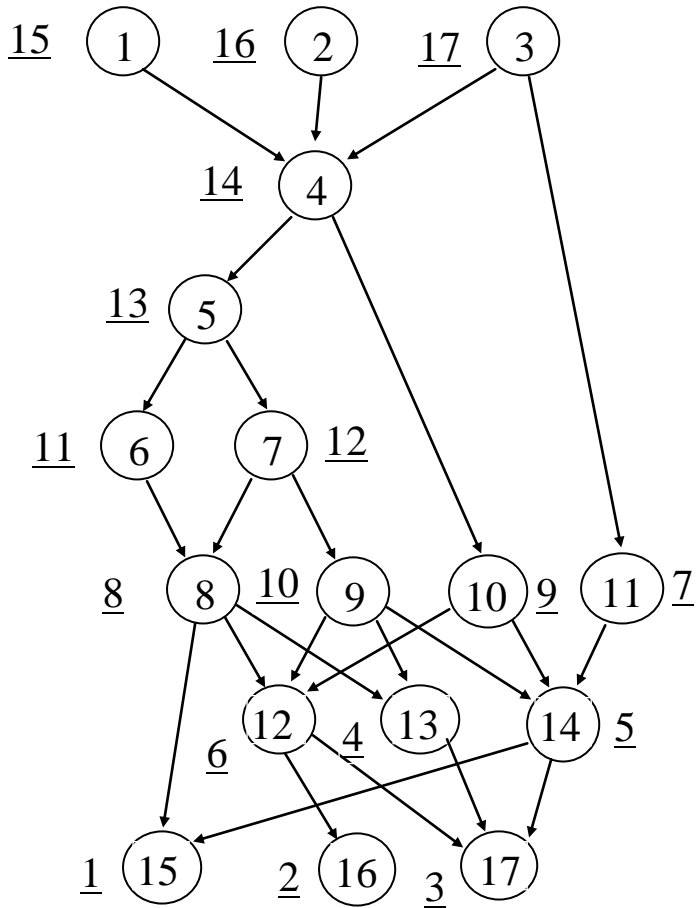
This is an optimal static algorithm that works on arbitrary task (program) graphs. Since the problem in general is NP-complete then some simplifying assumptions must be made. These assumptions are:

- 1) All tasks have the same execution time.
- 2) Communications is negligible in comparison to computation. Precedence ordering still remains.

The Algorithm

- 1) Assign labels $1, \dots, t$ to the t terminal tasks.
- 2) Let labels $1, \dots, j-1$ be assigned, and let S be the set of tasks with no unlabeled successors. For each node x in S define $l(x)$ as the decreasing sequence of the labels of the immediate successors of x . Label x as j if $l(x) \leq l(x')$ (lexicographically) for all x' in S .
- 3) Assign the highest labelled ready task to the next available time slot among the two processors.

Example



Gantt Chart for 2 Processors.

P1	3	1	4	5	7	9	12	13	17
P2	2	11		10	6	8	14	16	15

Dynamic Job Scheduling Algorithms

Job scheduling refers to the assumption that each task is independent. In general, dynamic algorithms can be classified as:

- a) *source-initiative algorithms*, where the processor that generates the task decides which processor will serve the task, and
- b) *server-initiative algorithms*, where each processor determines which tasks it will serve.

Examples of source-initiative algorithms are random splitting, cyclical splitting, and join shortest queue. Examples of server-initiative algorithms are random service, cyclical servicing, serve longest queue and shortest job first.

Server-initiative algorithms tend to out-perform source-initiative algorithms, with the same information content if the communications costs are not a dominating effect. However, they are more sensitive to distribution of load generation, and deteriorate quickly when one load source generates more tasks than another. But in heavily loaded environments server-initiative algorithms dominate source-initiative algorithms.

Real-Time Programming in Multiprocessor Systems

In recent years more real-time systems are being implemented on multiprocessor systems because:

- a) having multiple processors can reduce the workload on each processor and allows it to respond to an event within the deadline, and
- b) should one or more processors fail the systems can reconfigure the program and allocate the tasks to new processors so that the deadline are still met.

Therefore the two main issues in multiprocessor real-time programming are:

- 1) Scheduling tasks to meet their deadlines.
- 2) Fault tolerance.

Scheduling in Real-Time Systems

The goal of scheduling in a *hard real-time system* is to guarantee that all critical task meet their deadlines and that as many as possible essential tasks meet their deadlines.

Scheduling in real-time systems can be either synchronous or asynchronous.

Synchronous Scheduling

Synchronous scheduling algorithms are static scheduling algorithms in which the available processing time is divided by hardware clock into intervals called *frames*. Into each frame a set of tasks are allocated which will be guaranteed to be completed by the end of the frame. If a task is too big to fit into a frame it is artificially divided into a set of highly dependent tasks such that the smaller tasks can be scheduled into the frames.

Asynchronous Scheduling

Asynchronous scheduling can be either static or dynamic. In general dynamic scheduling algorithms are preferred since static algorithms cannot react to a change in the state of the computing system such as a hardware or software failure in some subsystem.

Dynamic asynchronous scheduling algorithms in a hard real-time system must still guarantee that all critical tasks meet their deadlines under specified failure conditions. This generally means these tasks are scheduled statically and that replicates of these critical tasks are statically allocated to several processors and that the active state information of the task is also duplicated. In the event of a processor failure the state information is sent to a duplicate of the task and all further inputs are rerouted to the replicate task.

The essential tasks are scheduled dynamically. There are two approaches to dynamically scheduling essential tasks.

- 1) To dynamically schedule essential and non-essential tasks together where the optimisation criterion is to minimise the number of essential tasks that fail to make the deadline, or minimise the maximum deadline violation.
- 2) As each essential task occurs in the system, the existing schedule is checked to see if the new essential task can be added to the schedule so that it makes its deadline. If so then the application is informed that the deadline of the essential task is guaranteed. If adding the task to the existing schedule cannot be accomplished without causing a previously guaranteed essential task to miss its deadline, then the application is informed that the essential task cannot be on-line guaranteed.

Examples of dynamic server-initiated hard real-time scheduling algorithms are *minimum deadline first* where the task with nearest deadline is scheduled next, and *maximum laxity first* where each selects the task with the minimum spare time before its deadline ($\min_j(D_j - P_j)$).

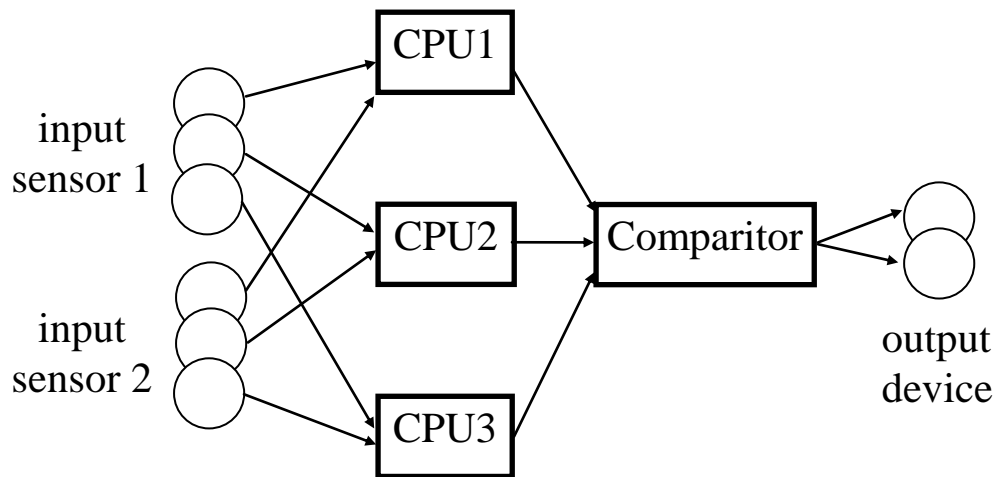
Fault Tolerant Concurrent Computing

As we saw earlier the main principles of fault tolerant programming were:

- 1) *Fault Detection* - Knowing that a fault exists.
- 2) *Fault Recovery* - Fault recovery depends on having atomic instructions that can be rolled back in the event of a failure being detected.

From a system's point of view it is quite possible that the fault is in the program that is attempting the recovery. Attempting to recover from a non-existent fault can be as disastrous as a fault occurring.

We have seen replication used at the task level to allow a program to recover from a fault which has caused a task to abruptly terminate. The same principle is also used at the system level to build fault tolerant systems. Critical systems are replicated, and the action taken by the system is based on a majority vote of the replicated sub-systems. This redundancy allows the system to successfully continue operating when several sub-systems develop faults.



Typical fault tolerant architecture

However, not all sub-systems fail gracefully. Sometimes, when a sub-system fails, it does not cease functioning; instead it continues to operate but generates incorrect data. This type of problem is called the Byzantine Generals problem.

Byzantine Generals Problem

This is a generalisation of the situation in which faulty processes are actively traitorous and send messages to other processes with the intent of causing a system failure.

A set of units from the Byzantine army are preparing to enter a battle. Each unit is led by a general,

and all generals communicate with each other by sending messengers. These messengers:

- i) Do not alter a message once it is given to them.
- ii) Always make to their destination.
- iii) Always identify the sender of the message.

The generals have pre-arranged a set of alternative actions, such as attack, retreat, or hold. The goal is to develop an algorithm such that:

- 1) All loyal generals take the same decision.
- 2) Every loyal general must base his decision on correct information from every other loyal general.

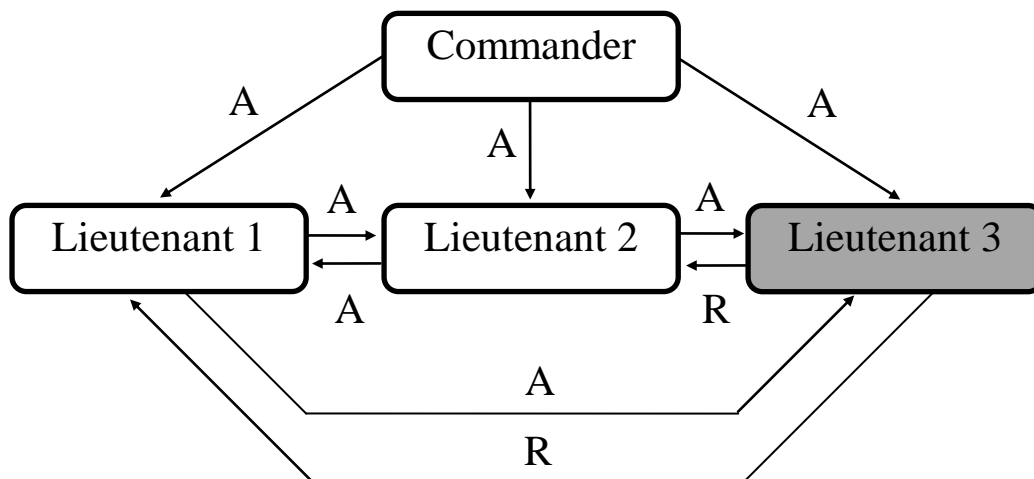
The Byzantine General Algorithm for One Traitorous General

One general, *the commander*, decides on an initial decision. The remaining generals are called *lieutenants*. The algorithm for one traitorous general is:

1. Commander sends his decision.
2. Each lieutenant relays the commander's decision to every other lieutenant.

- Upon receiving both the direct message from the commander and the relayed messages from the other lieutenants, the lieutenant decides on an action by majority vote.

If one of the lieutenants is the traitor, then each loyal lieutenant will receive $(n-3)$ correct messages from other loyal lieutenants, a correct message from the commander, and an incorrect message from the traitor. In order for there to be a majority n must be greater than 3. There is no known solution for only 3 generals.

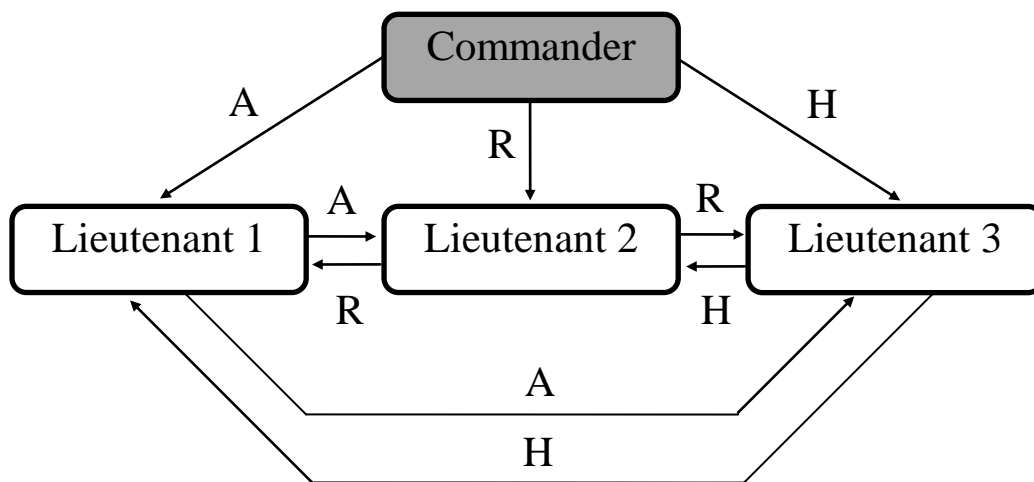


A \Rightarrow Attack

R \Rightarrow Retreat

A Traitorous Lieutenant

If the commander is the traitor, then it does not matter what messages he sends to the lieutenants since they are all loyal and will faithfully relay the message received from the commander. Each lieutenant will receive the exact same set of messages. Since lieutenants all react the same way on the information they receive, they will all make the same decision.



A \Rightarrow Attack

R \Rightarrow Retreat

H \Rightarrow Hold

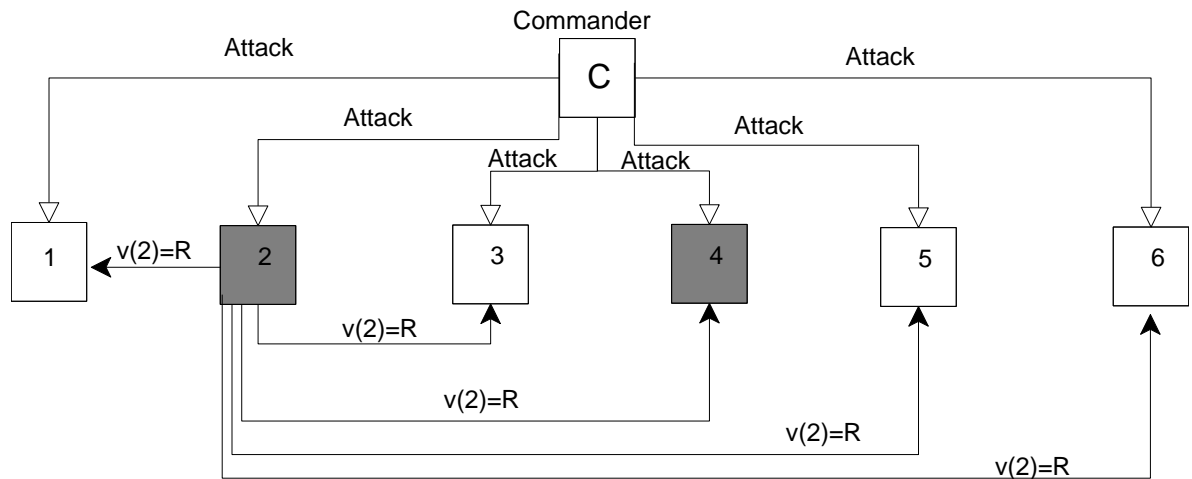
A Traitorous Commander

The Byzantine General Algorithm for Two Traitorous Generals

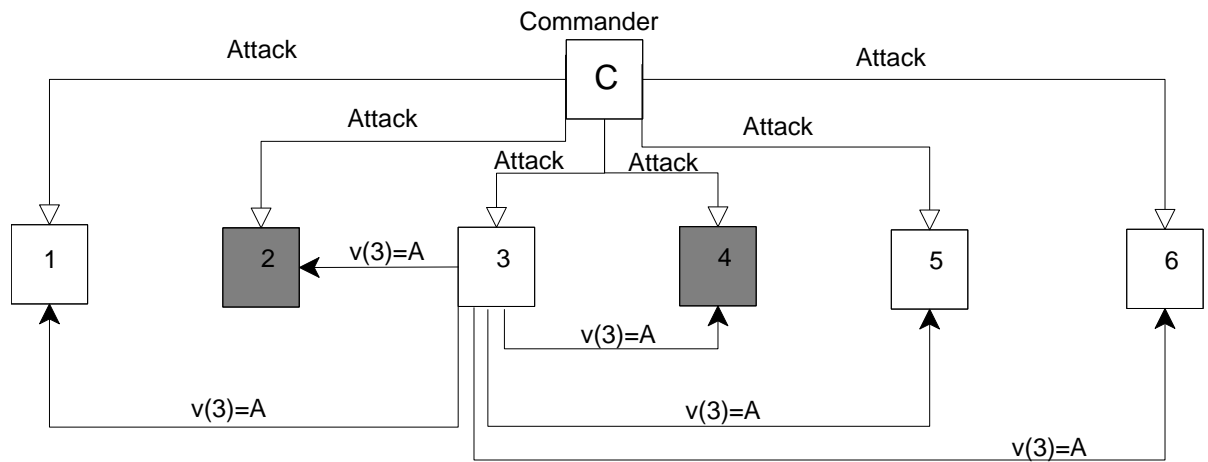
1. The commander sends his decision to each of the $n-1$ lieutenants. This is called the level-2 message.
2. Each lieutenant i sends the level-2 message to each of the $n-2$ other lieutenants. This is a level-1 message $v(i)$.
3. Each lieutenant k sends each level-1 message $v(i)$ to the other $n-3$ lieutenants. This is a level-0 message $v(i,k)$.
4. Eventually each lieutenant i receives $n-2$ messages from lieutenant j ; one level-1 message $v(j)$ and $n-3$ level-0 messages $v(j,k)$. Using majority vote lieutenant i can determine a value for lieutenant j .
5. Using the $n-2$ values from the other lieutenants and the level-2 message from the commander (commander could be traitor, after all), lieutenant i can use majority voting to determine his action.

In general an algorithm exists if less than a third of the generals are traitorous.

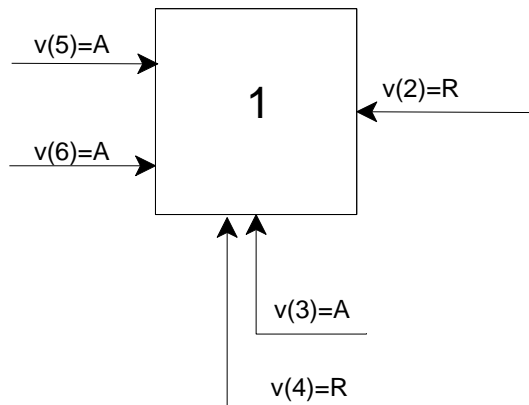
Two Traitorous Generals: Level 1 and Level 2 Messages Sent by Traitor and Commander (resp):



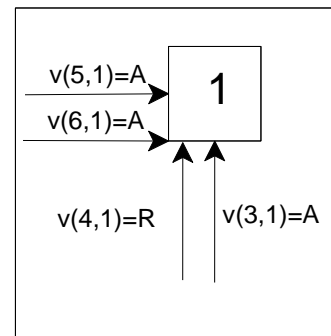
Two Traitorous Generals: Level 1 and Level 2 Messages Sent by Loyal Lieutenant and Commander (resp):



Level 1 Messages Received by 1



Level 0 Messages Sent by 1 to 2



What messages does (e.g.) Lieutenant 3 get:

Level/From	1	2*	-	4*	5	6
Level 0	RARA	RAAA	-	RAAA	RARA	RARA
Level 1	A	R	-	R	A	A
Majority Decision on others	A	A		A	A	A
Level 2	A					

Exercise:

Show that the above algorithm is correct for seven generals with two traitors. Show that the algorithm fails for only six generals with two traitors.

FIN