Embedded Systems 21

REVIEW: The Partitioning Problem

The **partitioning problem** is to assign
n **objects** $O=\{o_1, \ldots, o_n\}$ to
m **blocks** (also called **partitions**) $P=\{p_1, \ldots, p_m\}$
such that

- $p_1 \cup p_2 \ldots \cup p_m = O$
- $p_i \cap p_j = \emptyset$ for all $i \neq j$, and
- cost $c(P)$ is minimized.

REVIEW: Partitioning Methods

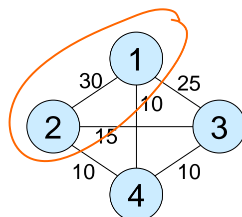
- **Heuristic methods**
  - Constructive methods
    - Random mapping
    - Hierarchical clustering
  - Iterative methods
    - Kernighan-Lin Algorithm
    - Simulated Annealing
- **Exact methods**
  - Enumeration
  - Integer Linear Programming (ILP)

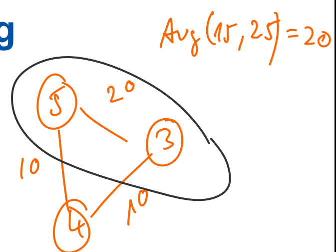
REVIEW: Hierarchical Clustering

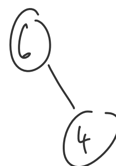
Avg (15, 25) = 20

Average closeness;

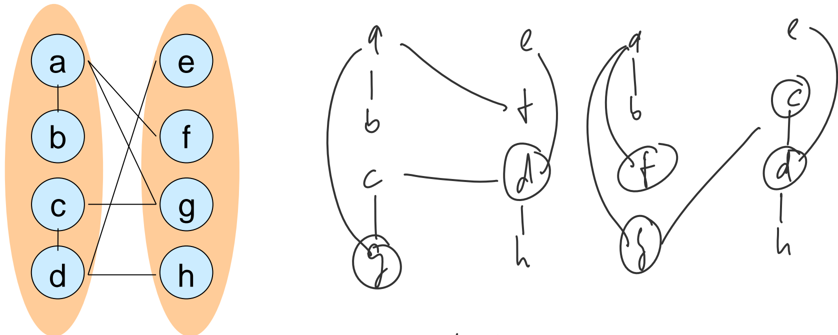
Termination: 2 blocks





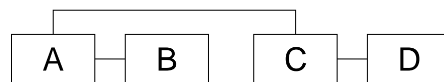


REVIEW: Kernighan-Lin



Step #	object pair	cut reduction	cut
0			5
1	{d, g}	3	2
2	{c, f}	1	1 ← least cut
3	{b, h}	-2	3
4	{a, e}	-2	5

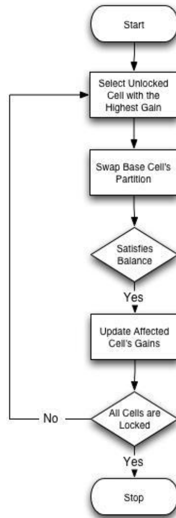
REVIEW: Fiduccia-Mattheyses Heuristic (F-M)



- Objects have size s(o)
- Size of block: sum of size of objects
- **Balanced two-way partition**:
  Given a fraction r, $0 < r < 1$,
  partition a graph into two blocks A and B such that
  $|A| / (|A|+|B|) \approx r$
  and cutset is minimized
- **Linear complexity**

Terminology: object=„cell“, hyperedges=„net“

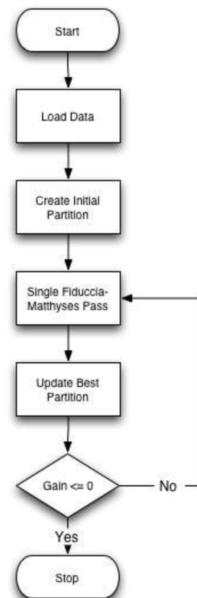
REVIEW: Single pass of the F-M heuristic



- Select the cell with the greatest gains that satisfies balance conditions
- Move the cell and lock it
- Update gains
- Repeat until all cells are locked or will dissatisfy balance conditions

REVIEW: Overall F-M heuristic

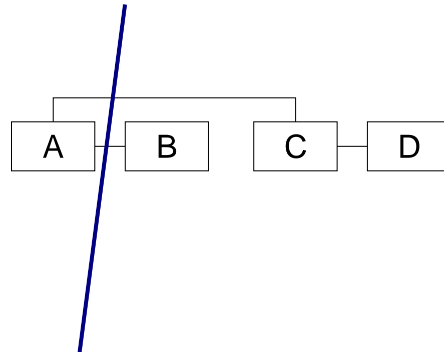
- Create an initial partition
- Execute a pass of the F-M heuristic
- Start again using the resulting partition as the initial partition
- Continue until the resulting gain is no longer greater than zero



REVIEW: Calculate gain

- g(i) = FS(i) - TE(i)
- FS(i): The number of nets which contain cell i but no other object in the same partition as i
- TE(i): The number of nets that consist only of i and other cells currently in the same partition as i

object	FS	TE	gain
A	2	0	2
B	1	0	1
C	1	1	0
D	0	1	-1



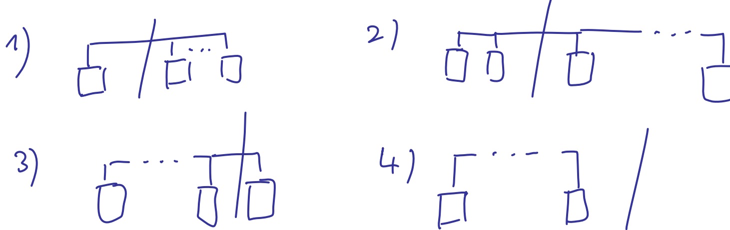
Calculate Gain

- **For** each unlocked cell i **do**
  - g(i) = 0
  - F = the “from block” of object i
  - T = the “to block” of object i
  - **For** each net n that contains i **do**
    - If F(n) = 1 increment g(i)
    - If T(n) = 0 decrement g(i)

Net distribution and critical nets

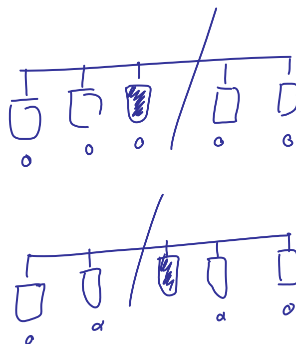
- Distribution of net i:
  (A(i),B(i))=(# of cells in A, # of cells in B)

- A net is **critical** if it has an cell that if moved will add or remove the net from the cutset
- Gain of a cell depends only on its critical nets.
- 4 cases: A(i)=0 or 1, B(i)=0 or 1

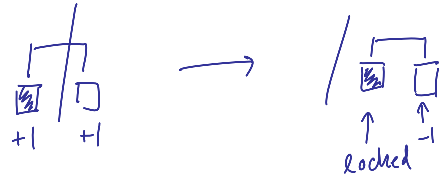


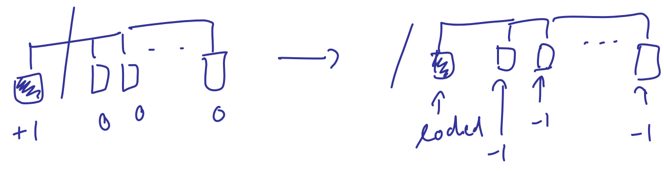
Updating gains

- For the update of the gains, we only need to consider nets that contain the cell selected for movement and that are critical before or after the move.

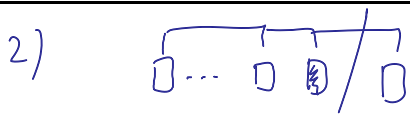


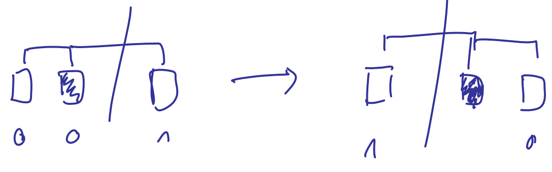
1)

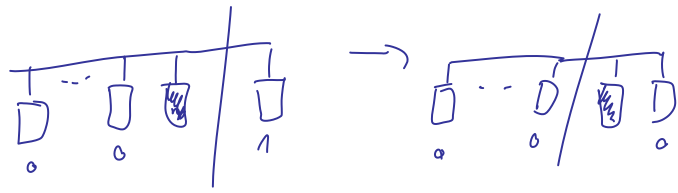




2)







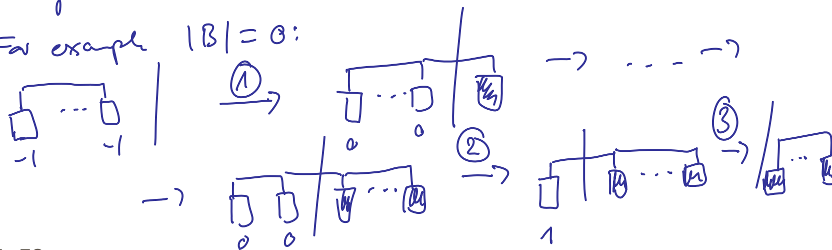
Case 3 / 4 analogously.

Complexity

- Once a net has some locked cell at both sides, the net will remain in the cut set.
- At most 3 update operations per net during one pass of the algorithm
- **Linear complexity**

Similarly for $|B|=1$, $|B|>1$

Wlog 1st move $A \to B$



Simulated Annealing

- General method for solving combinatorial optimization problems.
- Based the model of slowly cooling crystal liquids.
- Changes leading to a poorer configuration (with respect to some cost function) are accepted with a certain probability.
- This probability is controlled by a temperature parameter: the probability is smaller for smaller temperatures.

Simulated Annealing Algorithm

```
procedure SimulatedAnnealing;
var i, T: integer;
 begin
   temp := temp_start;
   cost:=c(P);
   while (Frozen()==FALSE) do
    begin
     while (Equilibrium()==FALSE) do
       begin P' := RandomMove(P);
         cost'=c(P')
         deltacost := cost' - cost;
         if (Accept(deltacost, temp)>random[0,1))
         then P=P'; cost=cost'
       end;
      temp:= decreaseTemp(temp)
   end;
end;
```


Simulated Annealing

- Annealing schedule: DecreaseTemp(), Frozen()
  - temp_start=1.0
  - temp = $\alpha \cdot$ temp (typical: $0.8 \leq \alpha \leq 0.99$)
  - stop at temp < temp_min or if no more improvement
- Equilibrium:
  - After certain number of iterations or when no more improvement
- Complexity:
  - From exponential to constant, depending on choice of Equilibrium(), DecreaseTemp(), Frozen()
  - The longer the runtime, the better the results
  - Usually functions constructed to obtain polynomial runtime

Integer programming models

- Ingredients:
- Cost function
- Constraints

Involving linear expressions over *integer* variables from a set X

Cost function $$C = \sum_{x_i \in X} a_i x_i \text{ with } a_i \in R, x_i \in \mathbb{N} \quad (1)$$

Constraints: $$\forall j \in J : \sum_{x_i \in X} b_{i,j}\, x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \quad (2)$$

Def.: The problem of minimizing (1) subject to the constraints (2) is called an **integer programming (IP) problem**.

If all x_i are constrained to be either 0 or 1, the IP problem said to be a **0/1 integer programming problem**.

Example

$$C = 5x_1 + 6x_2 + 4x_3$$

$$x_1 + x_2 + x_3 \geq 2$$

$$x_1, x_2, x_3 \in \{0,1\}$$

x_1	x_2	x_3	C
0	1	1	10
1	0	1	9
1	1	0	11
1	1	1	15

← Optimal (row with $C = 9$)

Remarks on integer programming

- Integer programming is NP-complete.
- Running times depend exponentially on problem size, but problems of >1000 vars solvable with good solver (depending on the size and structure of the problem)
- The case of $x_i \in \mathbb{R}$ is called *linear programming* (LP). LP has polynomial complexity, but most algorithms are exponential, still in practice faster than for ILP problems.
- The case of some $x_i \in \mathbb{R}$ and some $x_i \in \mathbb{N}$ is called *mixed integer-linear programming.*
- ILP/LP models can be a good starting point for modeling, even if in the end heuristics have to be used to solve them.

Integer Linear Programming for Partitioning

- Binary variables $x_{i,k}$
  - $X_{i,k}$=1: object o_i in block p_k
  - $x_{x,k}$=0: object o_i not in block p_k
- Cost $c_{i,k}$ if object o_i in block p_k
- Integer linear program:

$$X_{i,k} \in \{0,1\} \qquad 1 \le i \le n, 1 \le k \le m$$

$$\sum_{k=1}^{m} X_{i,k} = 1 \qquad 1 \le i \le n$$

minimize $\sum_{k=1}^{m} \sum_{i=1}^{n} X_{i,k} \cdot c_{i,k}$

A comprehensive integer linear programming model for HW/SW partitioning

Marwedel Section 6.3.2

Notation:

- Index set V denotes tasks.
- Index set L denotes task **types**
  e.g. square root, DCT or FFT
- Index set M denotes hardware component **types**.
  e.g. hardware components for the DCT or the FFT.
- Index set J of hardware component instances
- Index set KP denotes processors.
  All processors are assumed to be of the same type

An ILP model for HW/SW partitioning

- $X_{v,m}$: =1 if node v is mapped to hardware component type $m \in M$ and 0 otherwise.
- $Y_{v,k}$: =1 if node v is mapped to processor $k \in KP$ and 0 otherwise.
- $NY_{l,k}$ =1 if at least one node of type l is mapped to processor $k \in KP$ and 0 otherwise.
- $Type$ is a mapping from tasks to their types:
  $Type : V \rightarrow L$
- The cost function accumulates the cost of hardware units:

  C = cost(processors) + cost(memories) +
  cost(application specific hardware)

Constraints

Operation assignment constraints

$$\forall v \in V : \sum_{m \in M} X_{v,m} + \sum_{k \in KP} Y_{v,k} = 1$$

All task graph nodes have to be mapped either in software or in hardware.

Variables are assumed to be integers.

Additional constraints to guarantee they are either 0 or 1:

$$\forall v \in V : \forall m \in M : X_{v,m} \leq 1$$

$$\forall v \in V : \forall k \in KP : Y_{v,k} \leq 1$$

Operation assignment constraints (2)

$$\forall l \in L, \forall v{:}Type(v)=c_l, \forall k \in KP : NY_{l,k} \geq Y_{v,k}$$

- For all types l of operations and for all nodes v of this type:
  if v is mapped to some processor k, then that processor must implement the functionality of l.

- Decision variables must also be 0/1 variables:

$$\forall l \in L, \forall k \in KP : NY_{l,k} \leq 1.$$

Resource & design constraints

- $\forall\ m \in M$, the cost for components of type m is = sum of the costs of the components of that type.

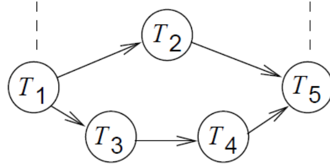
- $\forall\ k \in KP$, the cost for associated data storage area should not exceed its maximum.

- $\forall\ k \in KP$ the cost for storing instructions should not exceed its maximum.

Scheduling / precedence constraints

- For all nodes v_{i1} and v_{i2} that are potentially mapped to the same processor or hardware component instance, introduce a binary decision variable $b_{i1,i2}$ with
  $b_{i1,i2}$=1 if v_{i1} is executed before v_{i2} and
  = 0 otherwise.
  Define constraints of the type
  (end-time of v_{i1}) $\leq$ (start time of v_{i2}) if $b_{i1,i2}$=1 and
  (end-time of v_{i2}) $\leq$ (start time of v_{i1}) if $b_{i1,i2}$=0
- Ensure that the schedule for executing operations is consistent with the precedence constraints
- Approach fixes the order of execution
- Timing constraints guarantee that deadlines are met.

Example



- HW types $H1$, $H2$ and $H3$ with costs of 20, 25, and 30.
- Processors of type P.
- Tasks $T1$ to $T5$.
- Execution times:

T	$H1$	$H2$	$H3$	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

Operation assignment constraints (1)

T	$H1$	$H2$	$H3$	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

$$\forall v \in V : \sum_{m \in KM} X_{v,m} + \sum_{k \in KP} Y_{v,k} = 1$$

$X_{1,1}+Y_{1,1}=1$ (task 1 mapped to $H1$ or to P)
$X_{2,2}+Y_{2,1}=1$
$X_{3,3}+Y_{3,1}=1$
$X_{4,3}+Y_{4,1}=1$
$X_{5,1}+Y_{5,1}=1$

Operation assignment constraints (2)

- Assume types of tasks are l =1, 2, 3, 3, and 1.

$\forall\, l \in L,\ \forall\, v{:}Type(v)=c_{\,l},\ \forall\, k \in KP : NY_{\,l,k} \geq Y_{v,k}$

$$NY_{1,1} \geq Y_{1,1}$$
$$NY_{2,1} \geq Y_{2,1}$$
$$NY_{3,1} \geq Y_{3,1}$$
$$NY_{3,1} \geq Y_{4,1}$$
$$NY_{1,1} \geq Y_{5,1}$$

Functionality 3 to be implemented on processor if node 4 is mapped to it.

Other equations

- Time constraints leading to: Application specific hardware required for time constraints ≤ 100 time units.

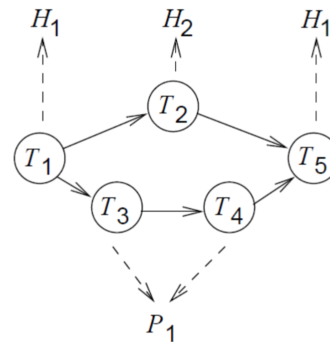
T	$H1$	$H2$	$H3$	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

Cost function:
C=20 #($H1$) + 25 #($H2$) + 30 # ($H3$) + cost(processor) + cost(memory)

Result

- For a time constraint of 100 time units and cost(P)<cost($H3$):

T	$H1$	$H2$	$H3$	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100



Solution (educated guessing) :
$T1 \rightarrow H1$
$T2 \rightarrow H2$
$T3 \rightarrow P$
$T4 \rightarrow P$
$T5 \rightarrow H1$

Fault tolerance

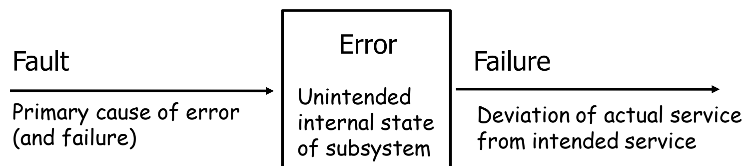
Safety vs. Reliability

- Safe means *sufficiently low probability of serious harm caused by the system:*
  - e.g. ISO 8402: „State in which risk of harm (to persons) or damage is limited to an acceptable level.“
- *Reliable* means *sufficiently high probability of delivering intended service.*
  - Reliability is the probability of the system delivering the service it was designed for throughout the horizon, given
    - a defined temporal horizon
    - the operational conditions

Faults, Errors & Failures

Standardized terminology: J. C. Laprie (ed.) 1992,

„Dependability: Basic Concepts and Terminology“



Example - landing gear in an airplane

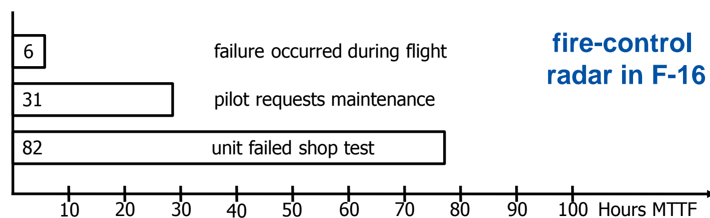
- Landing gear sensor faulty: doesn't report that gear is down
- Landing flaps and thrust-reverters are blocked by control software though plane is grounded
- Braking distance increases dramatically, plane may drive off runway

Dealing with Faults

- **Fault avoidance** aims at preventing the occurrence of faults: design reviews, testing, verification.
- **Fault tolerance** Is the ability of a system to continue to perform its tasks after the occurrence of faults
  - **Fault masking**: preventing faults from introducing errors
  - **Reconfiguration**: fault detection, location, containment and recovery

Types of faults

- A **permanent fault** remains in existence indefinitely if no corrective action is taken
- A **transient fault** disappears within a short period of time
- An **intermittent fault** may appear and disappear repeatedly.



fire-control radar in F-16

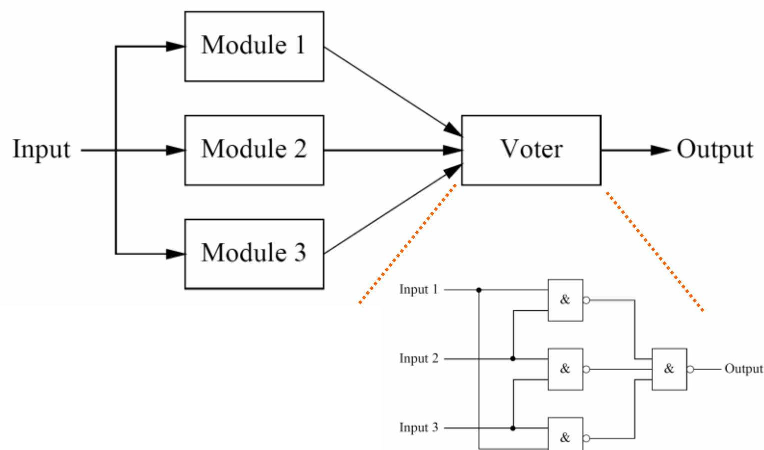
- Pilots noticed malfunctions every 6 flight hour
- Pilots requested maintenance every 31 hour
- Only 1/3 of the noticed malfunctions could be reproduced in the maintenance shop

Types of redundancy

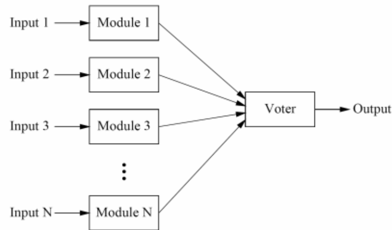
- **Hardware redundancy:** physical replication of hardware
- **Software redundancy:** different software versions of tasks, preferably written by different teams
- **Time redundancy:** multiple executions on the same hardware at different times
- **Information redundancy:** Coding data in such a way that a certain number of bit errors can be detected and/or corrected.

Static hardware redundancy

- Static redundancy based on voting.
- **Triple modular redundancy (TMR):**

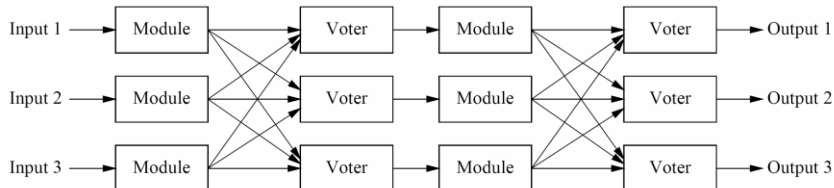


Static hardware redundancy: N-modular redundancy (NMR)

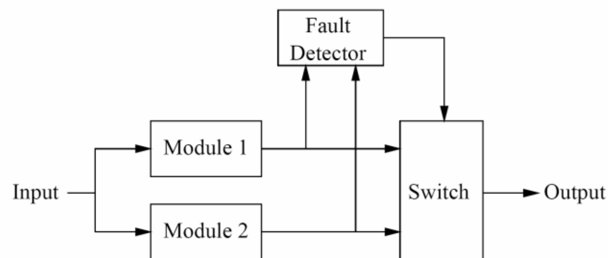


- System tolerates failure of (N-1)/2 modules
- Protects against random faults but not againts systematic faults
- Disadvantages: high cost, size, weight, energy. (typically: N≤4).

Static hardware redundancy: Multiple Stage TMR



Dynamic hardware redundancy: standby spare arrangement

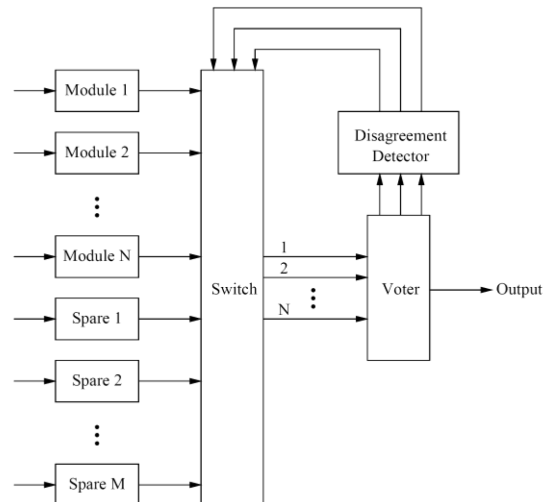


- Fault detection based on outputs (consistency check) not on voting
- Advantage: less redundant hardware
- Disadvantage: fault detection may take time ⇒ fault not masked

Standby spares

- **Hot standby:** spare is run continuously in parallel with active unit
  - Fast transfer of control
  - Increased power consumption
  - Same operating stress as active unit
- **Cold standby:** spare is unpowered until called into service
  - Reduces power consumption
  - Reduces wear and tear
  - More disruption at changeover

Hybrid redundancy: N-modular redundancy with spares



Software fault tolerance

- **N-version programming** (≈ static redundancy)
  - Prepare N different versions
  - Run them in parallel or sequentially
  - Select result of majority at the end
- **Recovery blocks** (≈ dynamic redundancy)
  - Each job has a primary version and one or more alternatives
  - When primary version is completed, perform acceptance test
  - If acceptance test fails, run alternative version

Danger: common-mode failures

- Ambiguities in specification
- Choice of programming language, numerical algorithms,…
- Common background of software developers

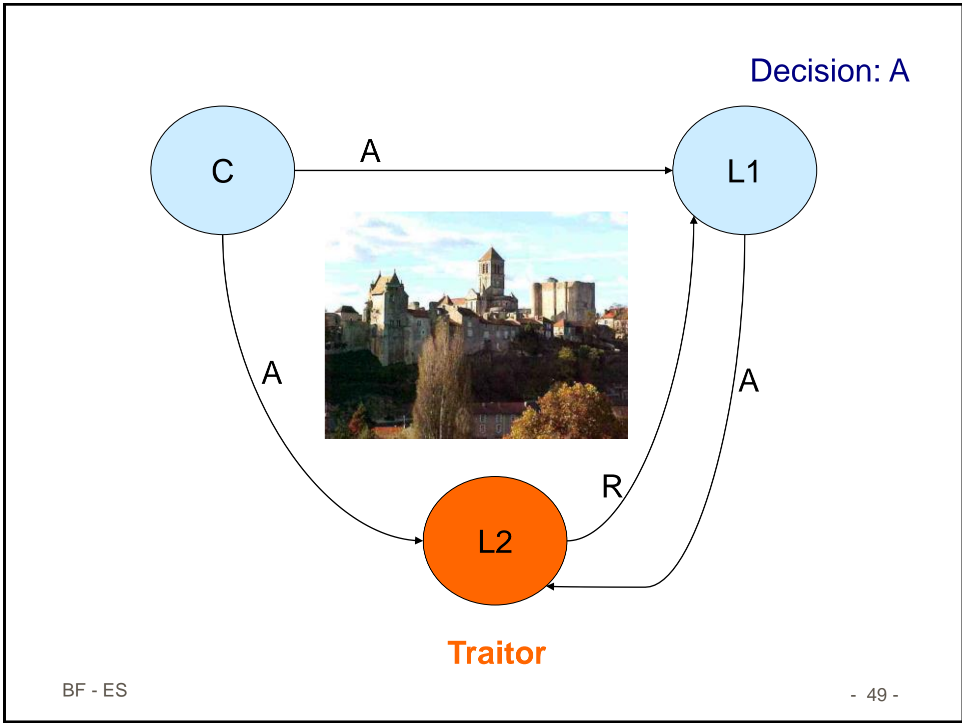
Failure modes of subsystems

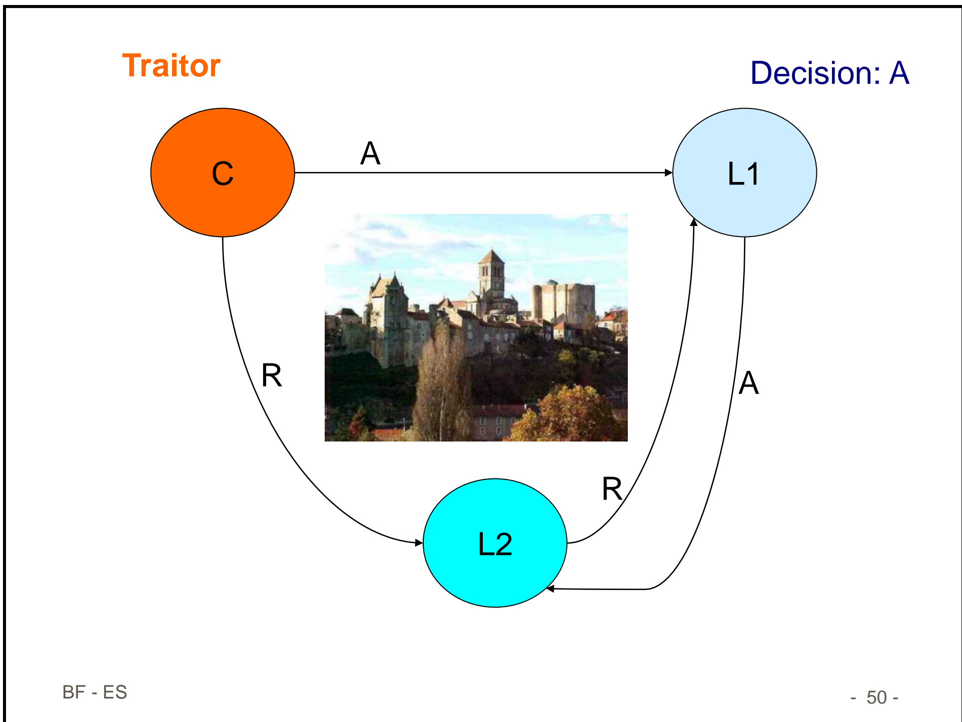
- Fail-silent failures
  - subsystem either produces correct results
    or produces (recognizable) incorrect results
    or remains quiet
  - **can be masked as long as at least one system survives**
- Consistent failures
  - If subsystem produces incorrect results all recipients receive same (incorrect) result
  - **can be masked iff the failing systems form a minority**
- Byzantine failures
  - subsystem reports different results to different dependent systems
  - **can be masked iff strictly less than a third of the systems fail**

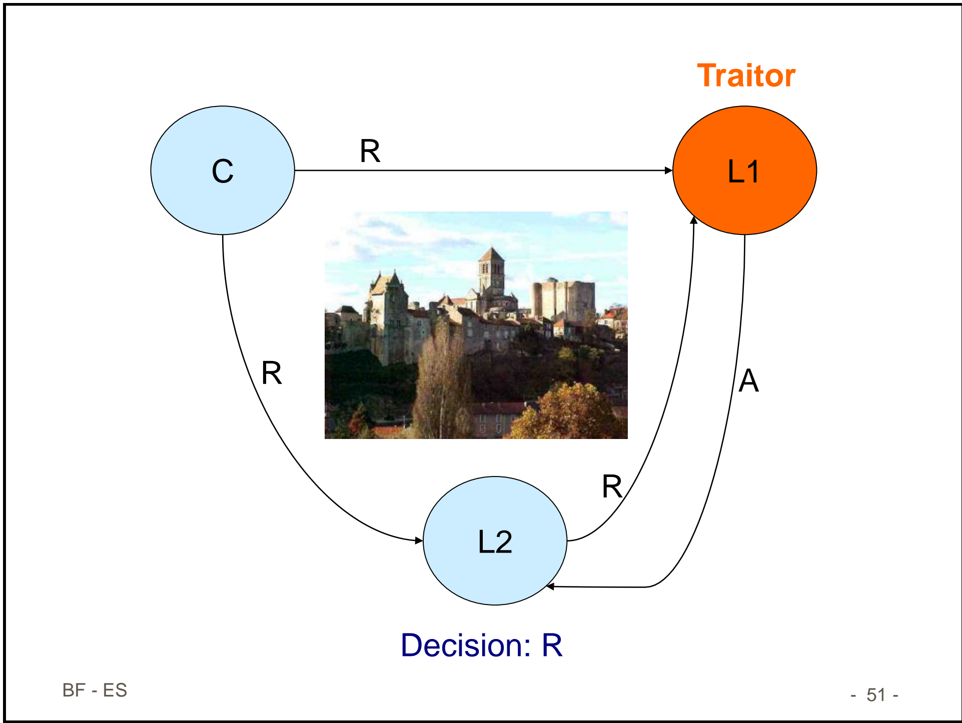
Byzantine generals [Lamport/Shostak/Pease´82]

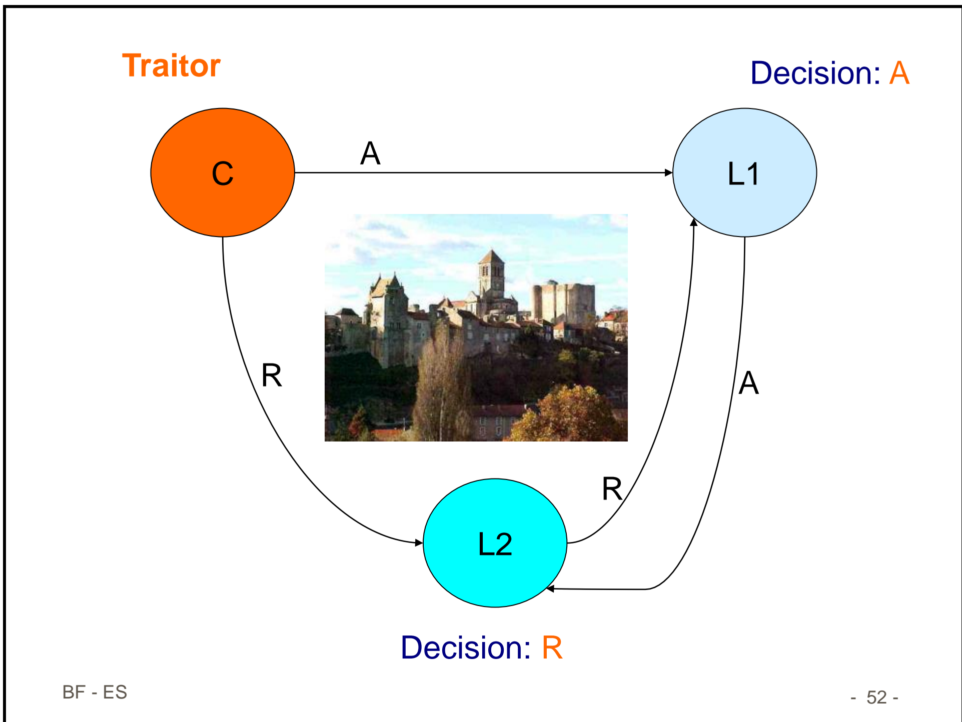
- Several divisions of the Byzantine army are camped outside an enemy city
- Each division is commanded by a general: there is one „commander“ and several „lieutenants“
- Each general may be a traitor
- Communication is reliable

- **Goal:** All loyal divisions must decide upon the **same** plan of action; if commander is loyal, loyal lieutenants should execute his order
- Basic idea: every lieutenant reports about the command received


Decision: A
C
A
L1
A
A
R
L2
Traitor
BF - ES
- 49 -


Traitor
Decision: A
C
A
L1
R
A
R
L2
BF - ES
- 50 -


Traitor
C
R
L1
R
A
R
L2
Decision: R
BF - ES
- 51 -


Traitor
Decision: A
C
A
L1
R
A
R
L2
Decision: R
BF - ES
- 52 -

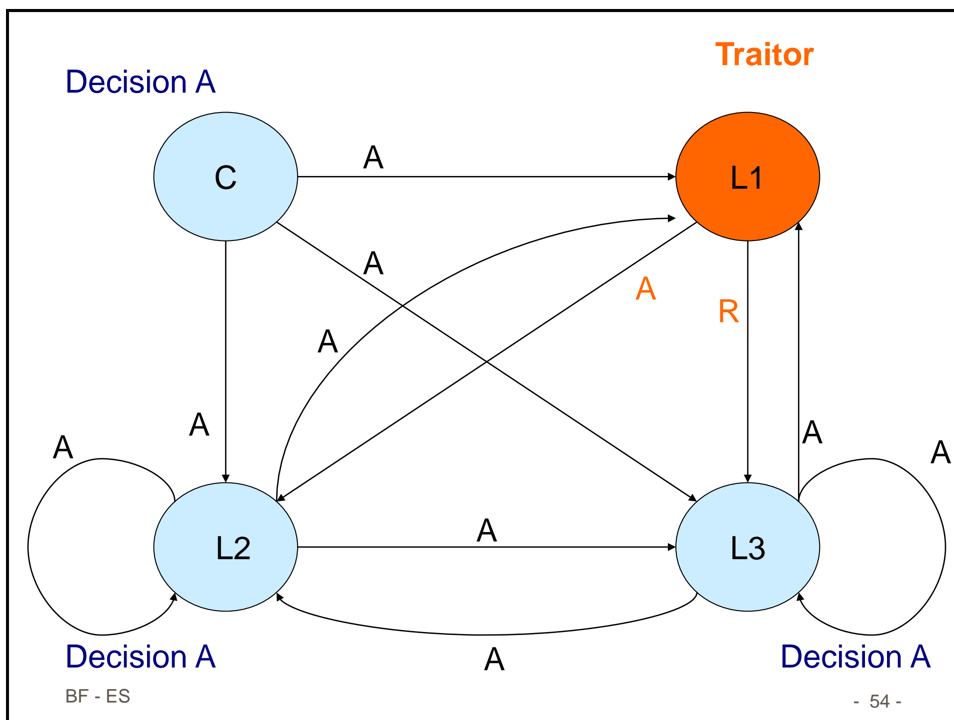
Solution

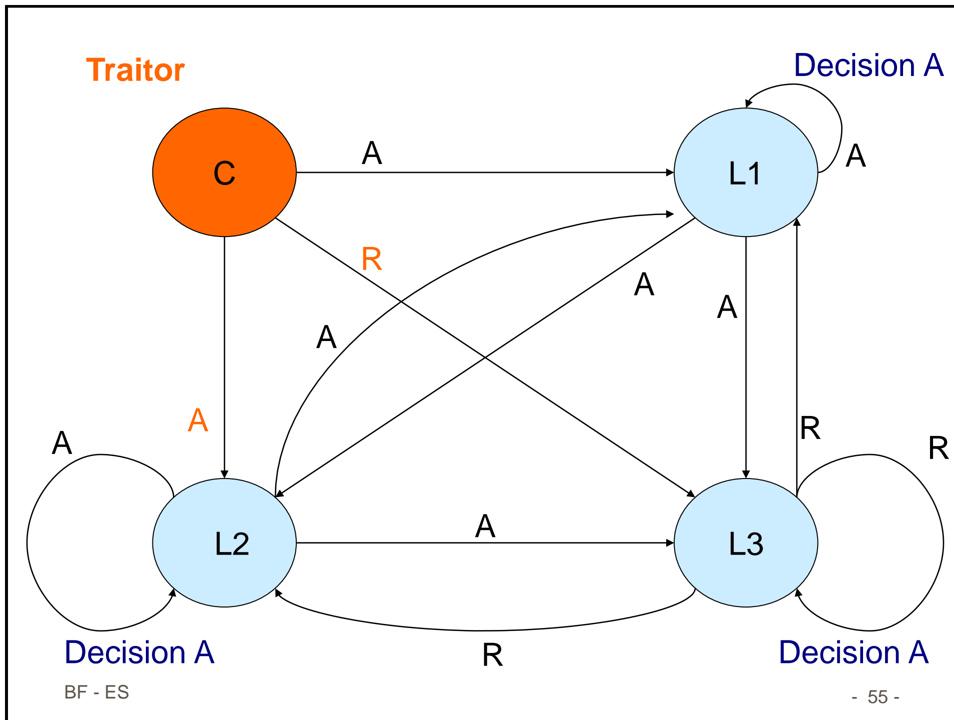
Algorithm A(0):

- Commander sends value (=order) to every lieutenant.

Algorithm A(*m*), *m*>0:

- Commander sends value to every lieutenant.
- Each lieutenant forwards value to all other lieutenants using algorithm A(*m*-1).
- Lieutenant *i* uses majority value of received values to determine result.





Lieutenants reach consensus (Case 1 traitor)

Case 1: Commander is loyal, some lieutenant is traitor

⇒ set of forwarded messages differ by at most 1 value

⇒ same majority vote

Case 2: Commander is traitor, lieutenants are loyal

⇒ sets of forwarded messages are identical

⇒ same majority vote at every lieutenant