Fault-Tolerant Systems

- Failure Semantics
- Fault-Tolerant software
- Fault-Tolerant hardware
- Fault-Tolerant data storage
  - Pessimistic solution — Stable storage
  - Optimistic solution — Replicated data and network partitioning
- Physical Clocks
  - MARS algorithm
- The most unreliable environment
  - Byzantine Generals Problem

Reliable Systems

- *reliability*
  - The probability that a system will perform in a correct manner
- *availability*
  - The probability that a system will deliver a correct answer at one moment
  - calculated from the system *reliability* and expected repair time after a fault.
- *fault-avoidance*
  - To try and make a perfect system
- *fault-tolerance*
  - To try and cope with errors using redundant parts or measures.

Fault-Tolerance

- A system might (for certain) contain faults.
  - Programming error
  - Hardware error
  - Data error
- When the execution of the system program hits a fault we will have an error.
  - Then the wanted output of the program can not be achieved in the normal manner.
  - If we have implemented measures to cope with this error we will still get a (almost) correct behavior of the system. This is called fault-tolerance.
    The error has been masked from the system behavior.
- When the system "user" sees the error or if the error leads to an inconsistent state or behavior then we have a failure.

Fault-Tolerance

methods

- redundancy
  - hardware
  - software
  - data
  - operation
- Different ambition
  - error detection and recovery
    - the errors are visible
  - error masking
    - the errors are hidden
    - gives transparency to the errors

Error Masking

- Hierarchical Error Masking
  - exceptions
  - message resend
  - retries
    - try to read from a hard disk a second time
  - ..
- Group Error Masking
  - group of servers
  - the MARS-algorithm
  - many copies of the same
    - Data File
    - Data Block

Model for error classification in Client-Server systems

- We have a
  - client

  that wants a
  - service

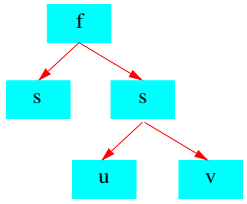
  which is carried out by a
  - server

  then the client is
  - dependent

  on that server.
- A server can in turn be using services from other servers.

  This can be described in a
  - dependency graph

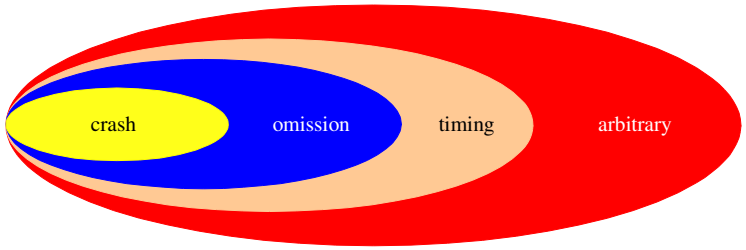
Dependency Graph



- f uses services from s and d.
- When carrying out f's service s uses services from u and v.

Failure Semantics

- Example (service):



- A crash error is when nothing happens.
- An omission error is when is when one or more responses fails. A crash is a special case of omission when all ropiness fails.
- A timing error is when is when one or more responses do not arrive in time. An omission error is a timing error when a response has infinite timing error.
- An arbitrary error is any error, i.e. a wrong value or a timing error.

Failure Semantics (2)

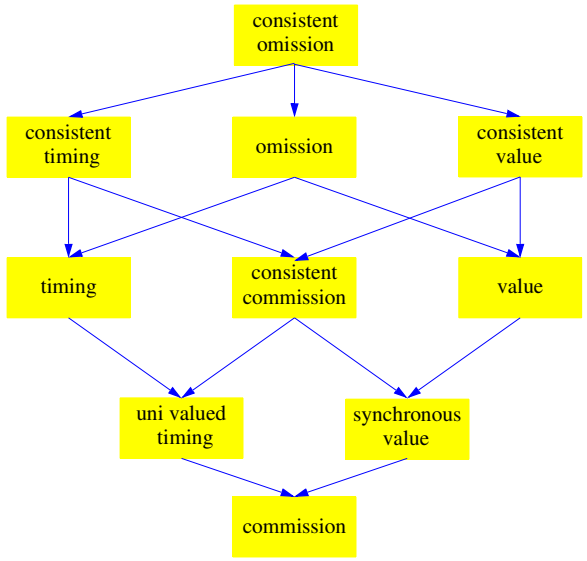
- When a client uses a server it can cope with different type errors from the server.
  - If it can manage a crash at the server it is said to assume the server to have crash failure semantics
  - If it can manage an service omission it is said to assume the server to have omission failure semantics
- Failure Semantics are the type of errors that we expect to appear.
  - Should another type of error appear it will lead to a service failure since we can not manage it.

Failure Semantics (3)

- The classifying depends on the performed algorithm.
- Example (Distributed Voting): (**NB** you don't have to learn this!)
  - *Omission Fault*
    A response fail to appear
  - *Consistent Omission Fault*
    All responses fail to appear, the same as a crash.
  - *Timing Fault*
    The response is not within the expected time interval.
  - *Consistent Timing Fault*
    All responses are not within the expected time interval.
  - *Univalued Timing Fault*
    All responses have the same *Consistent Timing Fault*.

- *Value Fault*
  A response has a wrong value.
- *Consistent Value Fault*
  All responses have wrong values.
- *Synchronous Value Fault*
  All responses have the same *Consistent Value Fault*.
- *Consistent Commission Fault*
  All responses have *Consistent Timing Fault* and *Consistent Value Fault*
- *Commission Fault*
  Responses have arbitrary faults, anything might be wrong in any manner.
  - arbitrary fault
  - Byzantine fault

- Example (Distributed Voting)



Choosing failure semantics

- At which level should the errors be treated?
- Which errors are allowed to cause a failure?
  - the different costs must be compared

Fault-Tolerant software

design diversity

- *N-version Programming*
- *Recovery Blocks*
- *Exceptions*

N-version Programming

- Write N different programs that performs the same task and compare each output value:
  - If the values are the same use that value
  - If the values differ and a majority of the programs has the same value: use that value
  - If there is no majority for one value the system fails.
- There must be at least 3 different programs implemented independently
  - Different implementing teams (companies)
  - Different implementation programs
  - Different implementing principles

  to avoid the same error to appear in the different programs.

N-version Programming **(cont.)**

- With a fail-stop system it is enough with 2 different programs
  - when the programs give different values the system will be stopped.
- Example: The Automatic Train Control System, ATC, for Swedish trains.
  - One program implemented in assembler language
  - One program implemented in Pascal
  - When output differs the train is emergency braked.
- This method can **not** be used for an airplane control system.

Recovery Blocks

- Make two different implementations of critical code.
- Use just one of them.
- Include a test for each output that it is within reasonable bounds.
  - If not, use the other implementation

Exceptions

- Test output values and use an exception if they are not within bounds.
- Catch exceptions and try to get a result in another way.

Fault-Tolerant hardware

- Use redundant hardware with the same software:
  - *comparing pairs*
    use two different hardware units and signal an error when values differs
  - *triple modular redundancy*, TMR
    use the majority value from three different hardware units
  - check pointing
    use tests within programs
  - *software acceptance test*
    test that the software has not been corrupted by using a checksum
  - *multiple units with voting*
  - *multiple units with reconfiguration*
  - *process migration*
    when a computer fails a process might be moved to another computer.
    e.g. TANDEM computers

Fault-Tolerant data storage

- Redundancy:
  - multiple copies of data objects
    - file level
      e.g. LOCUS multiple file copies
    - block level
      e.g. stable storage
    - disk level

    $\Rightarrow$ inconsistent data
  - we need atomic transactions

Stable Data Storage

- Redundant File System on the Block level - Lampson 1981.
- Instead of calculating the probability of success (which is impossible without reliable input data) we will instead show the different requirements for success.
- A system consisting of:
  - client programs
  - data storage units, *file servers*
  - address <f,b> corresponds to *byte* b on the file f
  - communication between client programs and *file servers*
- data storage units
  - *read*
  - *write*

  using messages

Stable Data Storage

- The Stable Data Storage is a layered architecture:

Stable Layer	Page copies			Group Error Masking
"Careful" Layer	"Careful" Storage	Crash - Restart	Ack -Resend	Hierarchical Error Masking
Physical Layer	Hard Disk	Processor	Network	

  - At the Physical Layer we give a model for the behavior of the units.
  - At the "Careful" Layer we use retry to take care of temporary errors.
  - At the Stable Layer we use multiple units to take care of more permanent errors.

The Physical System

Model

- Events

desired:	normal events	
error:	error events	foreseen, detected
disaster:	error events	**not** foreseen, undetected

The different algorithms will solve errors but not disasters.

- The physical units are:
  - secondary memory (hard disk)
  - data communication links
  - processors with primary memory

Hard Disks

- A Hard Disk consists of
  - a number of addressable pages
  - each page has a block of data and its status (*good*,*bad*)
  - the operations:
    - `procedure Put(at:Address;data:Dblock);`
    - `function Get(at:Address):(status:(good,bad);data:Dblock);`
- Two different types of event:
  - result of `Put` or `Get`
  - spontaneous events

Effects of the different operations

Get(A)			
	value of page A in the file	Get(A) returns	
desired	(good,d)	(good,d)	
desired	(bad,d)	(bad,...)	
error	(good,d)	(bad,...)	$\leq n_k$ times
disaster	(good,d)	(bad,...)	$> n_k$ times
disaster	(bad,d)	(good,...)	
disaster	(good,d)	(good,d')	d≠d'

Put(A,d)	
	page A becomes
desired	(good,d)
error	unchanged
error	(bad,d)

Spontaneous Events

decay

- Divide the units into pair of pages that are not decay related
  i.e. if there is damage to one page in the pair then the other page in the pair should be expected to be OK.
  - If there is a spontaneous change (damage) on a page in the pair then there should not be a spontaneous change on the other page in the pair within a time limit T_D.
    - Place the two pages in the pair on two different disk units.
      - damage to both within the time limit T_D will become a disaster
      - other damage will be expected errors

Spontaneous Events		
error	(good,d) → (bad,...)	only change within $[-T_D, T_D]$
error	(bad,d) → (good,d)	spontaneous correction
disaster	(good,d) → (bad,...)	several changes within $[-T_D, T_D]$
disaster	(s,d) → (s,d')	d≠d', undiscovered change

- We assume that addresses are not corrupted (otherwise disaster)
  according to the undiscovered change above.

Processors

- A processor error will be a crash

  ≡ *reset*
  - The processor state before the crash is lost

Communication

- messages:

  <status:(good,bad);data:Mblock;p:Processor>

- two operations:
  - procedure Send(to:Processor;data:Mblock);
  - function Receive():
    (status:(good,bad);data:Mblock);

Effects of the different operations

Receive()		
	The message is	Receive() returns
desired	<*good*,d,p>	(good,d)
desired	<*bad*,..,..>	(bad,null)

- We must allow a arbitrary delay on Receive

Send()		
	call	created message
desired	Send(to,d)	<good,d,to>

Spontaneous Events

- For communication all errors are spontaneous events

Spontaneous Events		
error	a message disappears	
error	a message is duplicated	
error	a message is damaged	<*good*,d,p>→<*bad*,d',p>
disaster	undiscovered damage	<*bad*,d,p>→<*good*,d,p>
	or	<*good*,d,p>→<*good*,d',p>d≠d'
	or	<*good*,d,p>→<*good*,d,p'>p≠p'

A Stable System

- First we define virtual units with less shortcomings than the physical units:
  - careful memory
- Operations:
  - *CarefulGet*
    Repeats *Get* until one page which is *good* is returned.
    At most n_k times.
  - *CarefulPut*
    Repeats *Put* and then *Get* to check the result until *Get* returns *good* and with same value as *Put* was supposed to write.
    At most n_k times.
- careful memory will mask occasional errors on a disk unit and transmission errors.


```
function CarefulGet(at:address):(Status,Dblock);
begin
 for i := 1 to n_k do
 begin
   (status,data) := Get(at);
   if status = good then return (status,data)
 end;
   return(status,data);
end;
```

```
procedure CarefulPut(at:Address;data:Dblock);
begin
 status := bad;
 i := 0;
 while ( status = bad ) and ( i < n_k ) do
 begin
   Put(at,data);
   (status,d) := Get(at);
   if d <> data then status := bad;
  i := i+1;
 end;
end;
```

Stable Storage

Consists of stable pages.

- Stable Page:
  - one data block
    (without status)
  - one ordered pair of *Careful*-pages that are **not** decay related.
    (every word in the definition is significant!)
- Operations:
  - *StablePut*
  - *StableGet*
  - *Cleanup*
- The value of a Stable Page will be:
  - the value of the first page in the pair if its status is *good*
  - otherwise the value of the second page in the pair without taking its status in account.

Stable Storage (cont.)

```
function StableGet(at:Address) : Dblock
 begin
   (ok,data) := CarefulGet(at copy 1);
   if not ok
      then (ok,data) := CarefulGet(at copy 2);
   return(data)
 end;
```

```
procedure StablePut(at:Address;data:Dblock);
 begin
   CarefulPut(at copy 1, data);
   CarefulPut(at copy 2, data);
 end;
```


Cleanup

```
procedure Cleanup;
 begin
   for all pages do
   begin
     (ok1,d1) := CarefulGet(at copy 1);
     (ok2,d2) := CarefulGet(at copy 2);
     if (ok1,d1) = (ok2,d2) then skip
     else if ok1 and not ok2 then
       CarefulPut(at copy 2,d1)
     else if not ok1 and ok2 then
       CarefulPut(at copy 1,d2)
     else if ok1 and ok2 then
       CarefulPut(at copy 2,d1)
     else skip /katastrof/;
   end;
 end;
```

- *Cleanup* is performed:
  - at system start
  - after each crash
  - at least every T_D seconds

Reliable Communication

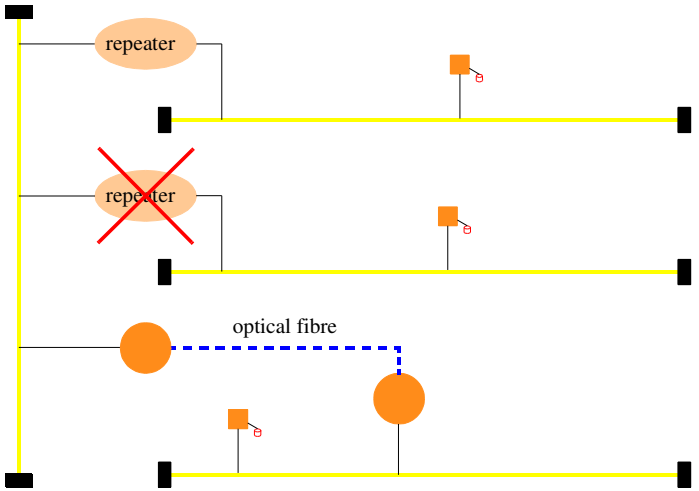
- Is achieved according to normal computer communication methods.:
  - computer communication protocol

Failure Semantics

hard disk:	*omission*	`Careful_Get`
	crash	`Stable_Get`

Network Partitioning

- replicated data
- consistent copies



Partitioning Graph

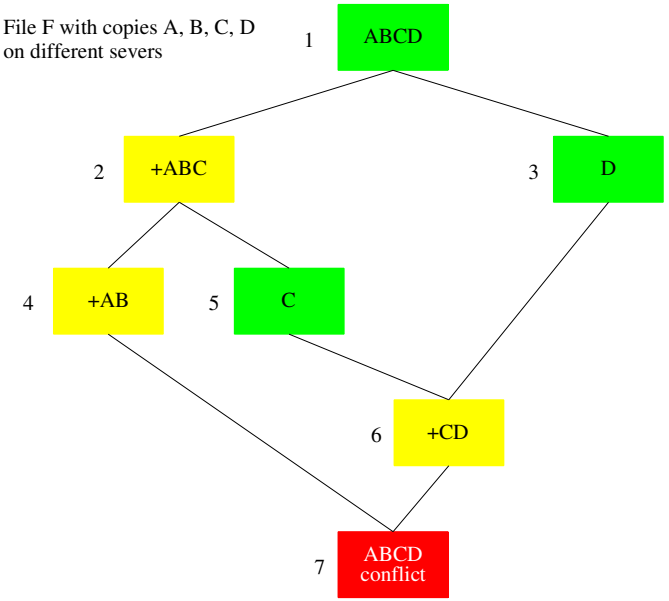
Definition: Partitioning graph G(f) for data unit f is a directed non-cyclic graph.

Each node gives the names of the units that belongs to the partition.

+ marks that the data f has been updated one or more times within the given partition.

Partitioning Graph example

File F with copies A, B, C, D
on different severs



Explanation of the Partitioning Graph

The Partitioning Graph for a file F describes which copies that are kept for the file and how the network might be divided into different partitions and if there is performed an update of the file within the partition. A + indicates that there has been at least one update within the partition.

- In our example we start with all copies in the same partition (1). This means that they can communicate with each other and that subsequently the files copies are equal.
- Then the network is divided into two partitions, 2 and 3. Copies A, B and C are in partition 2 and they perform an update which than is based on the file value in partition 1. Copies A, B and C will have the new value.
  Copy D is in partition 3 and keeps the value from 1.
- Then the network is divided again. A and B are in partition 4 while C is in partition 5.
  In partition 4 there is an update based on the value in partition 2. A and B gets the new value while C keeps the value from partition 2.
- Then part of the network is repaired so C and D gets connected in partition 6.
  Then D has the old value from partition1 while C has the value from partition 2.
  Since C's value is a newer value based on the value D has there is no real conflict,
  It is just to replace D's old value with C's new value and then they have the same value.
- Then the + for partition 6 indicates that an update. It is based on the value in partition 2.
- Then the network is completely repaired so all copy servers can communicate.
  Then they will have two conflicting values, both based on value in partition 2.

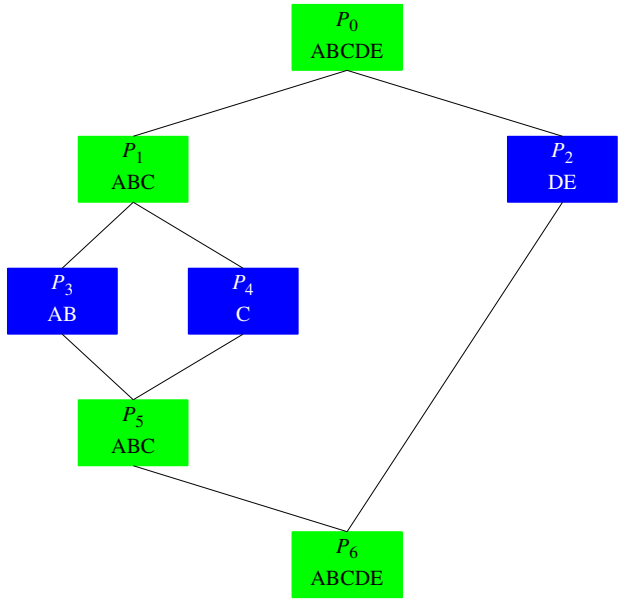
The main problem is to even avoid such conflicts to happen, pessimistic methods, or detect when they have happened.

Data Consistency (1)

- Pessimistic methods
  disallow concurrent updating/reading
  - primary copy
  - *token*
  - voting

  might give performance/availability problems.

Voting



Data Consistency (2)

- Optimistic methods
  - allow updates without locking
  - solve conflicts when partitions are merged
    reconciliation
  - *Roll Back*
    - cheap updates
    - expensive *reconciliation*
  - detection of conflicts:
    - Detects all possible conflicts without false alarms.
    - Version Vectors = Vector Clocks

Version Vectors

- Version Vector for a file:
  - a sequence of *n* pairs

    n number of copies of the file

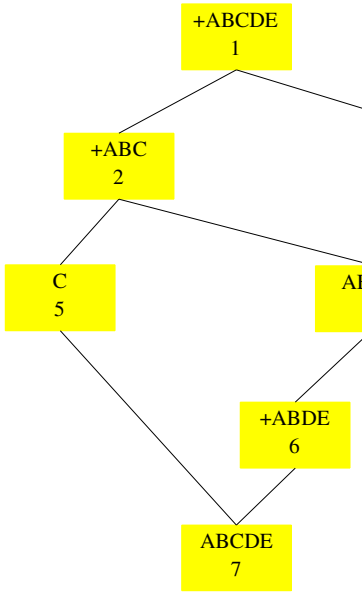
    pair *i*: (S_i : v_i) in the sequence indicates the number of updates that have been made at the node S_i keeping a copy of the file
  - Each copy of the file keeps a corresponding Version Vector.

  ⇒ Version Vectors are the same as Vector Clocks
- When updating a file;
  - Each update has a responsible node which has one of the file copies.
    The update is performed based on this file copy and the new version gets a Version Vector that is the old copy's Version Vector with the nodes own value interconnected by 1.
    Then all reachable copies of the file get the new version of the file together with the new Version Vector.
  - Multiple updates within one partition is not allowed, there can only be conflicts between updates in different partitions.

Version Vectors - Conflict Detection

- When two partitions, with version vectors V_i and V_j, are merged the following might hold:
  - $V_i = V_j$ in all positions:
    then no change has been made in any of the partitions. No action has to be taken.
  - $V_i < V_j$ according to the vector clock definition:
    then there has been at least one change in partition with vector V_j. The copies in the other partition should then be replaced with the latter version and they will also get its version vector.
  - $V_i \,//\, V_j$ according to the vector clock definition:
    then there has been at least one change in both partitions. This means that we have a conflict that must be solved to get a resulting version in the new partition.
    The new version will get the version vector where each position has the maximum value from the two old version vectors. Then one is added for the node that solved the conflict (calculated the new version), it actually does an update to the file.
    - Generally this conflict solving must be done manually. No algorithm can understand the content of a general file.
    - For a special file such as a directory the conflict can be solved automatically if for instance the two updates have been the creation of two files with different names.

Version Vectors — Example with no conflicts



partition	Version Vector
1	<A:1, B:0, C:0, D:0, E:0> A executes the update
2	<A:1, B:1, C:0, D:0, E:0> B executes the update
3	<A:1, B:0, C:0, D:0, E:0> no update, same as 1
4	<A:1, B:1, C:0, D:0, E:0> D,E gets same as 2
5	<A:1, B:1, C:0, D:0, E:0> no update, same as 2
6	<A:2, B:1, C:0, D:0, E:0> A executes the update
7	<A:2, B:1, C:0, D:0, E:0> C gets same as 6

1. In the first partition there is one update performed by A.
   All copies get the new value and the corresponding version vector will become <A:1, B:0, C:0, D:0, E:0> which is stored with all copies.

2. In this partition an update is performed by B. Copies A, B and C will be changed and get the version vector <A:1, B:1, C:0, D:0, E:0>.

3. In this partition there is no updates and the copies D and E will keep their values.

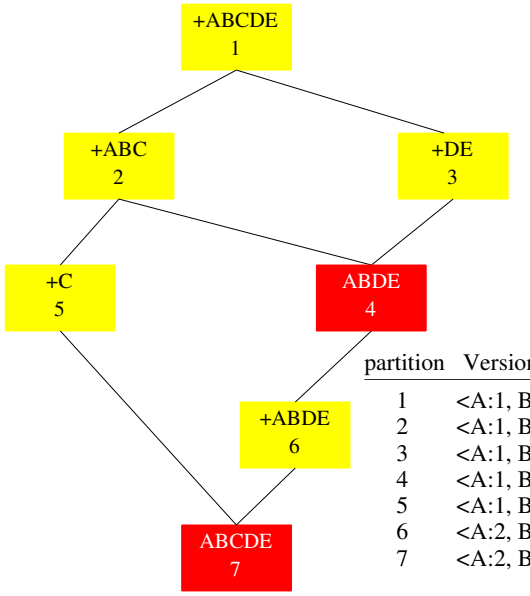
4. When merging copies A and B with D and E their version vectors are compared.
   It shows that the version vectors for D and E are strictly dominated by the version vectors of A and B. Thus D and E have older copies that not conflict with copies A and B. The older copies are changed to the new version and then also get the corresponding version vector.
   The version vector will become <A:1, B:1, C:0, D:0, E:0> for all copies.

5. In this partition there is no update.

6. In this partition an update is performed by A. Copies A, B, D and E will be changed and get the version vector <A:2, B:1, C:0, D:0, E:0>.

7. When C is merged with the others it has an older value. It gets the new value and version vector from the others.

Version Vectors — Example with conflicts



partition	Version Vector	
1	<A:1, B:0, C:0, D:0, E:0>	A executes the update
2	<A:1, B:1, C:0, D:0, E:0>	B executes the update
3	<A:1, B:0, C:0, D:1, E:0>	D executes the update
4	<A:1, B:2, C:0, D:1, E:0>	B resolves the conflict
5	<A:1, B:1, C:1, D:0, E:0>	C executes the update
6	<A:2, B:2, C:0, D:1, E:0>	A executes the update
7	<A:2, B:2, C:1, D:1, E:1>	E resolves the conflict

1. In the first partition there is one update performed by A.
   All copies get the new value and the corresponding version vector will become <A:1, B:0, C:0, D:0, E:0> which is stored with all copies.

2. In this partition an update is performed by B. Copies A, B and C will be changed and get the version vector <A:1, B:1, C:0, D:0, E:0>.

3. In this partition an update is performed by D. Copies D and E will be changed and get the version vector <A:1, B:0, C:0, D:1, E:0>.

4. When merging copies A and B with D and E their version vectors are compared.
   It shows that there is a conflict. The conflict is solved by B. The version vector for the new copies will have the maximum value in each position from vectors 2 and 3. Then the value for B is incremented since B resolves the conflict.
   The version vector will become <A:1, B:2, C:0, D:1, E:0> for A,B,D and E copies.

5. In this partition there is an update by C based on the copy from partition 2. The version vector will become <A:1, B:1, C:1, D:0, E:0> which is stored with the C copy.

6. In this partition an update is performed by A based on the copy from partition 4.
   The version vector will become <A:2, B:2, C:0, D:1, E:0> for A,B,D and E copies.

7. When C is merged with the others there is a conflict again. After solved by E the version vector for the new copy will be <A:2, B:2, C:1, D:1, E:1> which is stored with all copies.

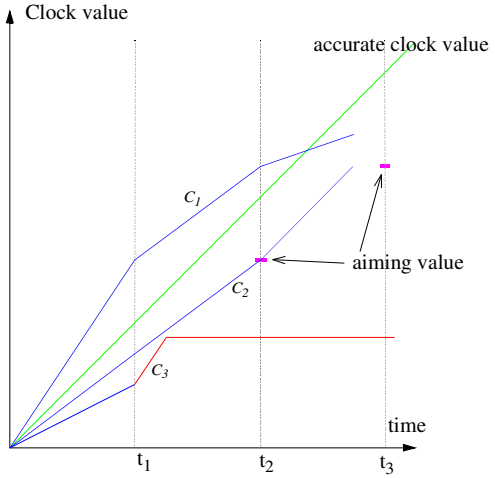
Version Vectors

- Version Vectors introduces high availability in case of network part since we are allowed to do updates even if we don't have access to all copies (one is enough!)
  - Good if we know that there is a low risk for simultaneously updates for the same file.
  - If there still would be simultaneously updates we are in trouble, but it will be **detected**.

Physical Clocks revisited

- Fault-tolerant version of the physical clock setting algorithm.
  - Send messages with physical clock value timestamp to each neighbor.
  - When receiving a message set the clock according to clock algorithm **but**:
    - Exclude the lowest and the highest value
    - MARS algorithm
- When a clock fails it will be too fast or too slow (or possibly within the right limit but then it would not be a problem!).
  - The algorithm excludes theses wrong values.

MARS algorithm



- Before calculating the mean value, the highest and the lowest value are excluded.
  This means that if at most one clock stops functioning that value will not be included.
  In the example the c_3 clock stops after time t_1. This will not affect the calculated value that will become the same as the c_2 value.

Byzantine Generals

- Coping with the most difficult types of faults

  e.g. “disloyalty” when performing mutual decisions
- Byzantine faults

  commission faults

  arbitrary faults
- The Byzantine Generals Problem:
  - A general send its orders to its *n-1* lieutenants

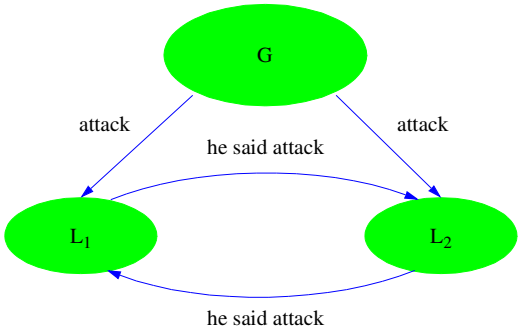
    (i) All loyal lieutenants should follow the same order.

    (ii) If the general is loyal, then all loyal lieutenants should follow his order.

Byzantine Generals

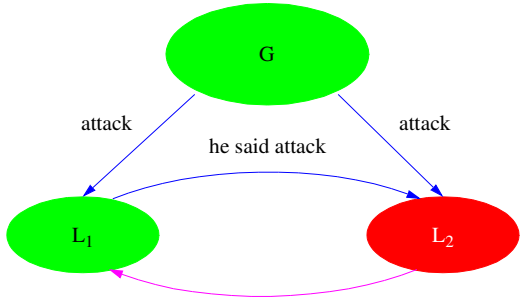
- To fulfill the requirement we must try to find the disloyal officer.
  - By comparing each other orders we might be able to follow an order according to the requirement by using a majority vote.
- How many officers do we need?
  - We try 3.

Byzantine Generals — three officers



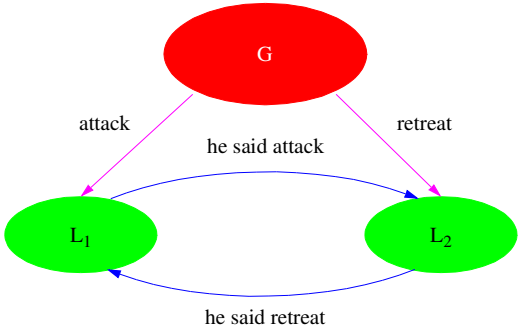
Loyal officers

Byzantine Generals — three officers



L_2 disloyal

Byzantine Generals — three officers

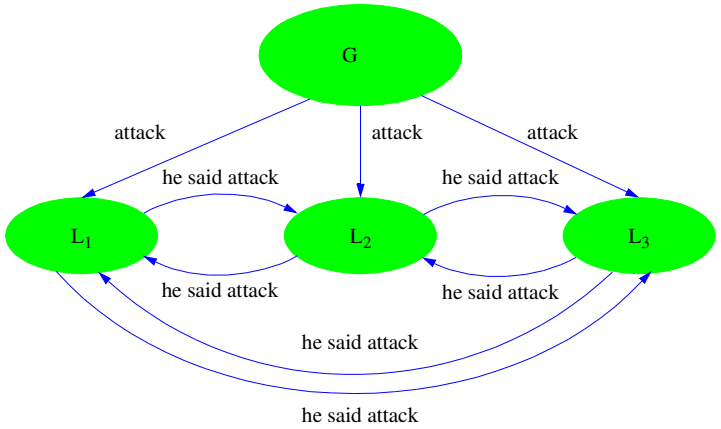


The general disloyal

Byzantine Generals — three officers

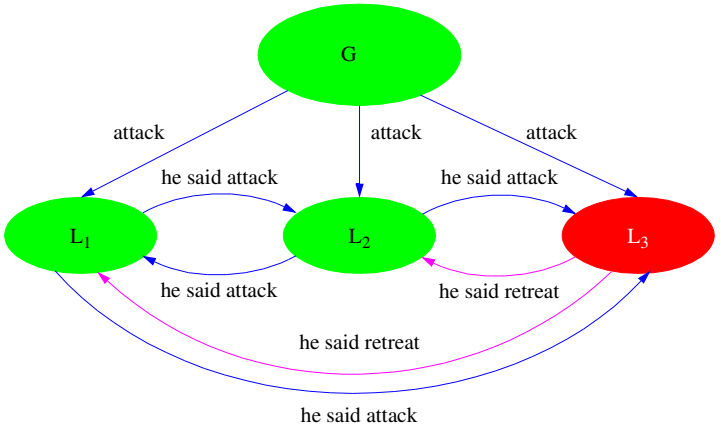
- We find that the officers can not follow the requirements!
  - L_1 doesn't know which order to follow.
- We try 4 officers instead.

Byzantine Generals — four officers



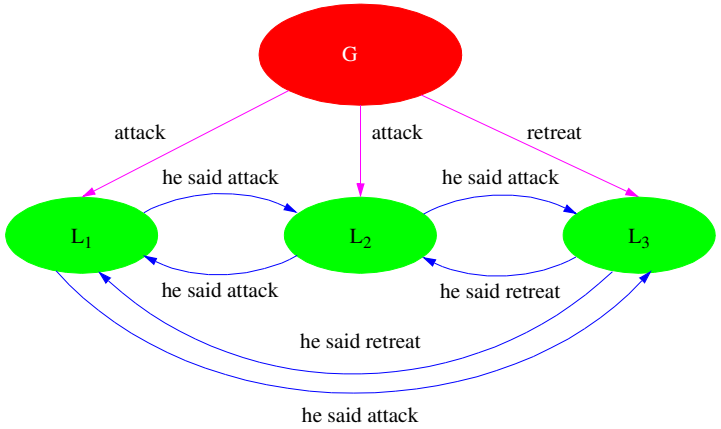
Loyal officers

Byzantine Generals — four officers



L_3 disloyal

Byzantine Generals — four officers



The general disloyal

Byzantine Generals — four officers

- Now the loyal officers will choose the same order and thus fulfill the requirements.

Byzantine Generals — Computer Applications

- Byzantine faults

  Requirements:

  - At most *m* of *3*m+1* nodes fail

- Voting
  - The process giving its vote is the General.
  - For each voting round each process then takes turn as General.
  - Used in the *Space Shuttle* control system:
    - 4 computers running the same program compares their results.
    - Will protect the shuttle from errors due to data change due to cosmic rays if there is only damage in one of the computers.
- The *Space Shuttle* also has a fifth computer with the control program written in another language.
  - This can be shifted to manually if there are doubts about the other program.
  - There can then be no shift back to the first program!
  - N-version programming with N = 2.