Pervasive Systems

Ioannis Chatzigiannakis

Sapienza University of Rome
Department of Computer, Control, and Management Engineering (DIAG)

Lecture 8:
Agreement in Distributed Computing

Hierarchical Analysis of Systems Performance

A fundamental method for studying the performance of a system is the top-down approach

- Initially we abstract all technical details and study the system at high level (i.e., bird's eye view)
- Then, we look into specific modes of operations and investigate the most important parameters that affect performance.
- Step - by step, we introduce additional levels – until we end up to our final system, operating in the actual conditions

This approach leads to good results for organizing and analyzing a broad range of systems

Contemporary Systems

Hierarchical, centralized, top-down approaches have allowed us to design very good contemporary systems

- e.g., database management systems, mobile telephony networks

However our always-connected world is becoming more complex

- We should not ignore the fact that many contemporary systems have a totally different structure.
- e.g., the stability and effectiveness of contemporary politico-economic models relies on decentralized, distributed mechanisms that are independent and self-regulated
- The Internet is another example of a similar approach, at a techno-social level.
- How to efficiently organized extremely huge collections of unstructured or structured data ?

Limitations of Top-down approach

Studying a pervasive system from a theoretical perspective contributes to a basic level of understanding its behavior and rigorously defining its performance bounds.

However, it entails certain pitfalls:

- Abstracting certain technical details may lead to totally unrealistic / non-implementable solutions.
- Measuring complexity does not take into account the so-called "hidden" constants.
- A "poor" complexity solution may be very efficient in almost all practical case.
- It is very hard (if not impossible) to analyze the performance of a system using theoretical tools.

Performance evaluation by Experimentation

A different approach is the implementation of the system and its evaluation using practical means:

- The implementation may use an experimental framework – e.g., simulator, testbed facilities, ...
- The performance study is done using well-defined evaluation scenaria
- Measure performance of the "actual" performance.
- Immediate validation of the applicability of a solution in existing technologies.
- Results can be deployed to devices in real-world deployments.

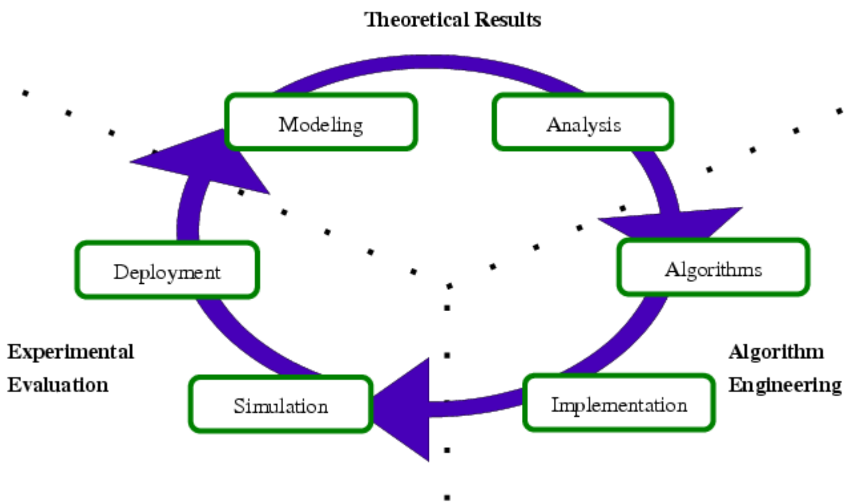
Dual Approach

Each approach has certain benefits and handicaps:

- A theoretical approach allows to develop solutions that are correct by proof, efficient ... may not be applicable (or very hard) in current technologies.
- A practical approach immediately deals with all technological issues and provides effective solutions ... may not result in innovative solutions that are efficient in large scale systems.

We need to be both efficient and effective.

Theoretical – Practical Approach Cycle



Necessity of Dual Approach

- Surprisingly, the need for association between theory and practice has been identified long before the computer science era.
- Philosophers of the antiquity, already, state that the notion of effectiveness requires two components: design an efficient prototype, an *ideal version* that is used to plan the goal; then, *apply* this plan in practice.
- According to Plato, $\nu o \eta \sigma \iota \varsigma$ (cognition) "captures optimal" plans, and $\theta \epsilon \lambda \eta \sigma \iota \varsigma$ (goodwill) is required to apply the ideal plans in reality.
- This is defined by Aristotle as $\phi \rho o \nu \eta \sigma \iota \nu$, the process of associating the ideal with its application, thus reducing the gap between these two approaches.

Modeling Processes

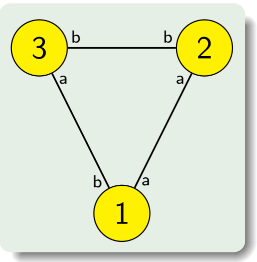
- The system is comprised from a collection of processing elements or "processors".
  - The "processing element" suggests a piece of hardware.
  - The "processors" suggests some kind of logical entity (i.e., software).
  - For simplicity we may assume that each processing element has 1 processor.
- Processors execute a collection of processes.
  - For simplicity we may assume that each processor executes only one process.
  - We also assume that each process can be executed by a single processor.

Modeling the Communication Network

- The processing elements (i.e., the processes) are connected via a connected network (i.e., there exists 1 path between any pair of processes).
- We define the network as a graph $G = (V, E)$:
  - comprised of a finite set V of points – the vertices – representing the processing untis (i.e., processes) – $n = |V|$
  - a collection E of ordered pairs of elements of V ($E \subset [V]^2$) – the edges – representing the communication channels of the network – $m = |E|$

Modeling Communication Channels

- Channels are the edges of the graph.
  - The edges may be directed – to represent unidirectional communication.
  - or undirected – to represent bidirectional communication.
- Processes can distinguish each communication channel and select a specific one to use.

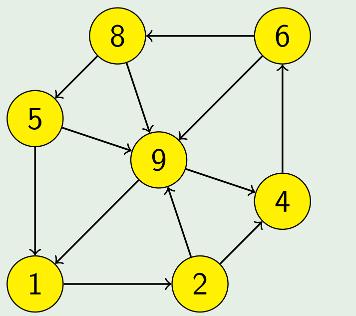


Modeling Messages

- Data exchange over communication channels is done via message exchanges.
- We assume that each communication channel may transmit only one message at any time instance.
- We assume that there exists a fixed message alphabet M
  - remains fixed throughout the execution of the system.
  - contains the symbol `null` a placeholder indicating the absence a message.

Neighboring Processes

- We say vertex v is outgoing neighbor of vertex u if
  - the edge uv is included in G.
- We say vertex u is incoming neighbor of vertex v if
  - the edge uv is included in G.
- We define $nbrs_u^{out} = \{v|(u,v) \in E\}$ all the vertices that are *outgoing neighbors* of vertex u.
- We define $nbrs_u^{in} = \{v|(v,u) \in E\}$ all the vertices that are *incoming neighbors* of vertex u.



5 is outgoing neighbor of 8
8 is incoming neighbor of 5
$nbrs_9^{out} = \{1,4\}$
$nbrs_9^{in} = \{2,5,6,8\}$

Network Properties

$\mathtt{distance}(u,v)$

Let $\mathtt{distance}(u,v)$ denote the length of the shortest directed path from u to j in G, if any exists; otherwise $\mathtt{distance}(u,v){=}\infty$.

$\mathtt{diam}(G)$

Let $\mathtt{diam}(G)$ denote the diameter of the graph G, the maximum distance $\mathtt{distance}(u,v)$, taken over all paths (u,v).

Network Topology & Initial Knowledge

- Distributed algorithms may be designed for a specific network topology
  - ring, tree, fully connected graph ...
- Distributed algorithm may be designed for networks with specific properties
  - we say that the algorithm has "initial knowledge"
- An algorithm assuming a large number of specific properties is called "weak" algorithm.
  - An algorithm that does not assume any specific property is called "strong" algorithm – since it can be executed in a broader range of possible networks.

Process States

- Each process $u \in V$ is defined by a set of states $states_u$
  - A nonempty set of states $start_u$, known as starting states or initial states.
  - A nonempty set of states $halt_u$, known as halting states or terminating states.
- Each process uses a message-generator function $msgs_u : states_u \times nbrs_u^{out} \rightarrow M \cup \{\mathtt{null}\}$
  - given a current state,
  - generates messages for each neighboring process.
- Uses a state-transition function $trans_u : states_u \times (M \cup \{\mathtt{null}\})^{nbrs_u^{in}} \rightarrow states_u$
  - given a current state,
  - and messages received,
  - computes the next state of the process.

System Initialization

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- Algorithms groups processes in two sets
  1. Initiators – a process is initiator if it activates the execution of the algorithm in the local neighborhood.
  2. Non-initiators – a non-initiating process is activated when a message is received from a neighboring process.

Centralized vs Decentralized

An algorithm is classified as centralized if the exists one and only one initiator in each execution and decentralized if the algorithm may be initialized with an arbitrary subset of processes.

- Usually centralized algorithms achieve low message complexity.
- Usually decentralized algorithms achieve improved performance in the presence of failures.

Uniformity

An algorithm is uniform if its description is independent of the network size n.

- A property that holds for a small network size, also holds for large network sizes.
- We only have to examine the behavior of a protocol (for a given property) in small network sizes.

Algorithm execution: Steps and Rounds

- All processes, repeat in a "synchronized" manner the following steps:

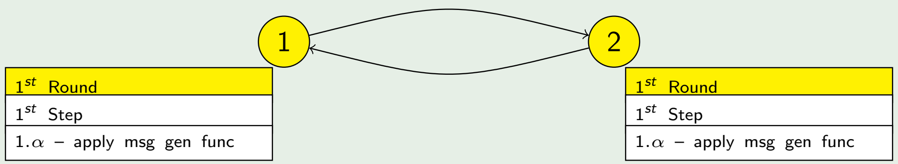
  1^{st} Step
  1. Apply the message generator function.
  2. Generate messages for each outgoing neighbor.
  3. Transmit messages over the corresponding channels.

  2^{nd} Step
  1. Apply the state transition function.
  2. Remove all incoming messages from all channels.
- The combination of these two steps is called a round (of execution).

Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

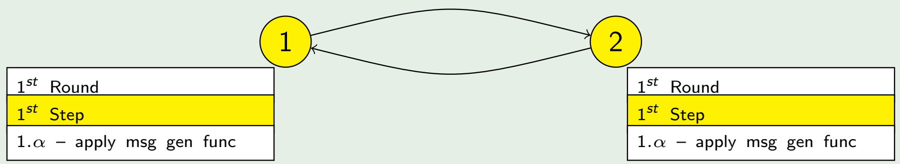
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

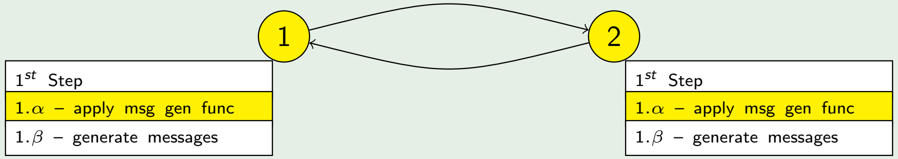
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

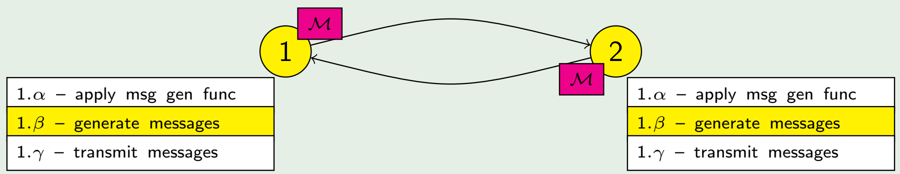
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

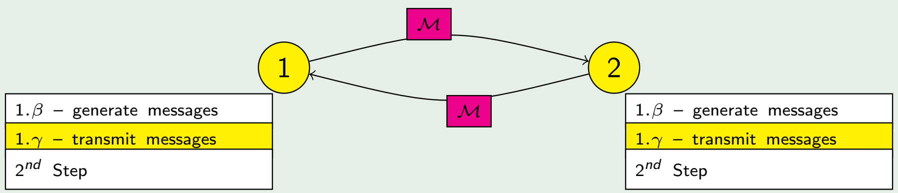
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

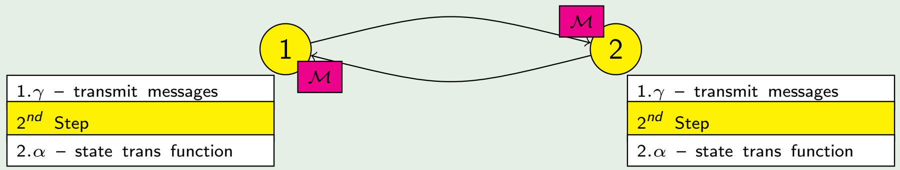
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

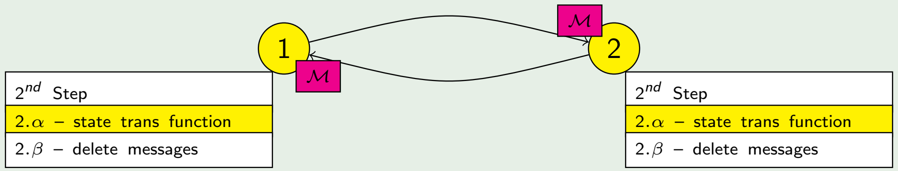
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

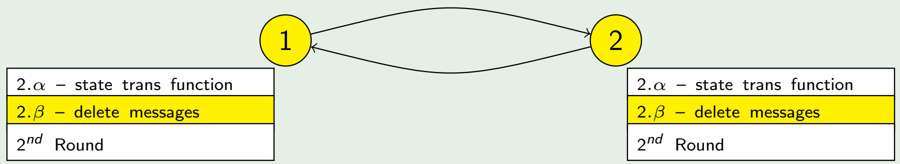
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

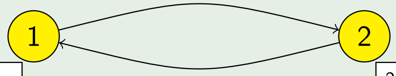
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

Execution of Synchronous System

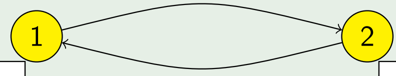


$2.\beta$ – delete messages	$2.\beta$ – delete messages
2^{nd} Round	**2^{nd} Round**
1^{st} Step	1^{st} Step

Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

Execution of Synchronous System

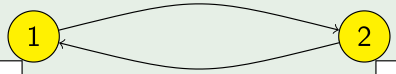


2^{nd} Round	2^{nd} Round
1^{st} Step	**1^{st} Step**
$1.\alpha$ – apply msg gen func	$1.\alpha$ – apply msg gen func

Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

Execution of Synchronous System

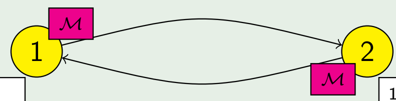


1^{st} Step	1^{st} Step
$1.\alpha$ – apply msg gen func	**$1.\alpha$ – apply msg gen func**
$1.\beta$ – generate messages	$1.\beta$ – generate messages

Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

Execution of Synchronous System

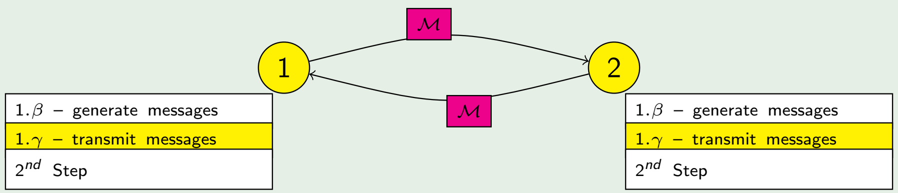


$1.\alpha$ – apply msg gen func	$1.\alpha$ – apply msg gen func
$1.\beta$ – generate messages	**$1.\beta$ – generate messages**
$1.\gamma$ – transmit messages	$1.\gamma$ – transmit messages

Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a “synchronized” manner the protocol.

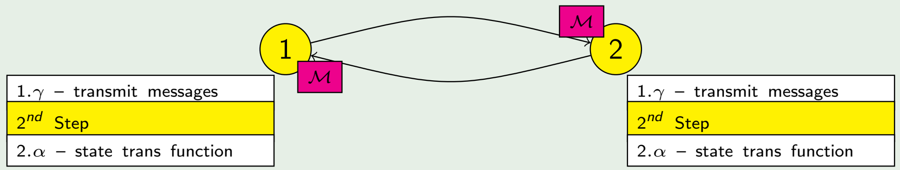
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a “synchronized” manner the protocol.

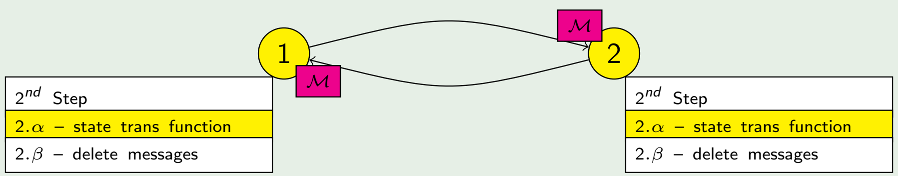
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a “synchronized” manner the protocol.

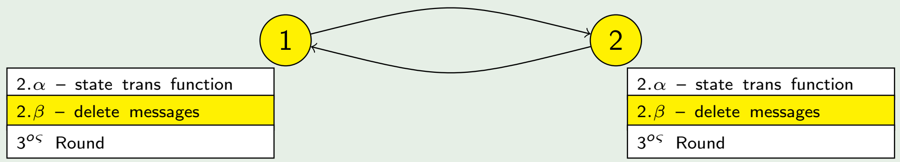
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a “synchronized” manner the protocol.

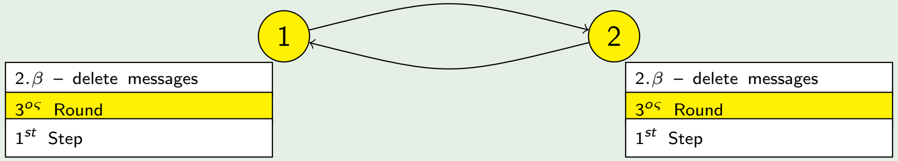
Execution of Synchronous System



Example of execution of a Synchronous System

- Initially
  - all processes are set to an initial state,
  - all channels are empty.
- the processes execute in a "synchronized" manner the protocol.

Execution of Synchronous System



System Configuration

We wish to describe the execution of a distributed algorithm.

- We assume a sequence of state transitions of the processes of the system
  - produced as result of transmissions and receptions of messages, or
  - internal (to each process) reasons.
- Lets assume a given time instance i
  - each process u is in state $states_u$.
  - the characterization of the state of all processes defines a configuration of the system C_i.

Execution of a distributed algorithm

- Initially, processes execute a single round of the algorithm
  - a given set of message transmissions M_i take place,
  - a given set of message N_i are received.
- The next round $i+1$, we say that the system is in configuration C_{i+1}
- The execution of the distributed algorithm can be defined as an infinite sequence $C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots$

Basic Failure Types

- We define two abstract types of failures:
  1. failures occurring during the transmission of messages,
  2. failures occurring on the processing elements (processors).
- Communication failure: a failure during the transmission of a single message over a specific channel of the network.
- Stopping failure: a process terminates, either before, or after, or during the execution of some part of the 1^{st} or 2^{nd} step of the round.
  - A failure may happen during the generation of messages, therefore some outgoing messages are transmitted.

Byzantine Failures

- The network includes faulty processes that do not terminate but continue to participate in the execution of the algorithm.
- The behavior of the processes may be completely unpredictable.
- The internal state of a faulty process may change during the execution of a round arbitrarily, without receiving any message.
- A faulty process may send a message with any content (i.e., fake messages), independently of the instructions of the algorithm.
- We call such kind of failures as Byzantine failures.
- We use byzantine failures to model malicious behavior (e.g. cyber-security attacks).

Why study Byzantine Fault Tolerance?

- Does this happen in the real world?
  The "one in a million" case.
  - Malfunctioning hardware,
  - Buggy software,
  - Compromised system due to hackers.
- Assumptions are vulnerabilities.
- Is the cost worth it?
  - Hardware is always getting cheaper,
  - Protocols are getting more and more efficient.

Measuring Performance

- We wish to study the performance of the system.
  - We define the minimum requirement,
  - Select a suitable distributed algorithm.
- How can we measure performance?
- We use to fundamental metrics to define the complexity of distributed algorithms:
  1. Time complexity
  2. Communication complexity

Time Complexity

The time complexity of a synchronous system is defined as the total number of rounds required for all the processes to produce all the necessary output, or until all processes enter a halting state.

- Directly related with the execution time of an algorithm.
- In practice, the execution time of a distributed algorithm is the most important performance metric.

Communication Complexity

The communication complexity of a synchronous system is defined as the total number of non-null messages exchanged during the execution of the system.

- In some cases it is measured in total number of `bits` exchanged.
  - in cases when the volume of messages produces congestion in the network,
  - and the execution of the algorithm is delayed (for the network to deliver messages).

Communication Complexity

- In real conditions
  - multiple algorithms are executed concurrently,
  - they share the same communication medium.
  - What is the contribution of each algorithm to the total network congestion ?
- It is difficult to quantify the effect that the messages of each algorithm have on the performance of the other algorithms.
- In general, at design time, we always wish to minimize the messages produced by our algorithms.

Books & Seminal Papers

1. Nancy A Lynch: "Distributed Algorithms". Morgan Kaufmann (1996)
2. Michael J. Fischer, Nancy A. Lynch: "A Lower Bound for the Time to Assure Interactive Consistency". Inf. Process. Lett. 14(4): 183-186 (1982)
3. Harry R. Lewis, Christos H. Papadimitriou: "Elements of the Theory of Computation". Prentice Hall (1981)
4. Barbara Liskov, Alan Snyder, Russell R. Atkinson, Craig Schaffert: Abstraction Mechanisms in CLU. Commun. ACM 20(8): 564-576 (1977)

Stopping Failures

Processes may simply stop arbitrarily without warning, at any point during a round of execution of a distributed algorithm. The process will halt immediately and terminate without further interaction with the other processes of the system.

- Stopping failures model unpredictable processor crashes.
- We assume an upper bound σ on the number of stopping failures
  - such an upper bound holds for the complete execution of the distributed system.
  - is equivalent to other measures, e.g., rate of stopping failure per round.

FloodSet Algorithm

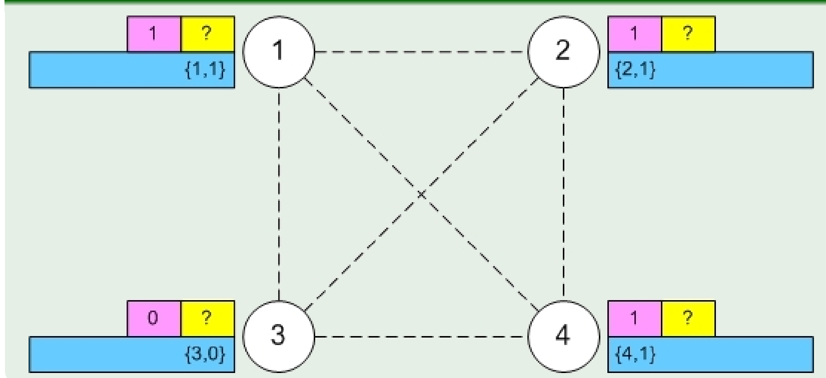
Each process $u \in [1, n]$ maintains a list I_u with input values, initially included only the input value $i_u \in S$ of u, $I_u = \{i_u\}$. In each round, each process broadcasts I, then adds all the elements of the received sets to I_u. After $\sigma + 1$ rounds, if I_u is a singleton set (i.e., $|I_u| = 1$), then u decides on the unique element of I_u; otherwise u decides on the default value $i_0 \in S$.

- We assume a complete graph G.
- We assume an upper bound on process failures σ
- Let $I_u(\gamma)$ be the values in I_u of u at round γ

Example of execution of FloodSet algorithm

Let a synchronous complete graph $n = 4$ and $\sigma = 2$.

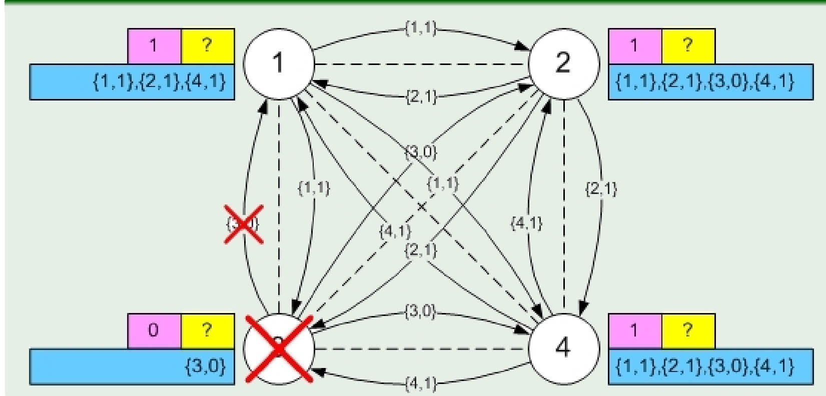
Complete Graph



Example of execution of FloodSet algorithm

Let a synchronous complete graph $n = 4$ and $\sigma = 2$.

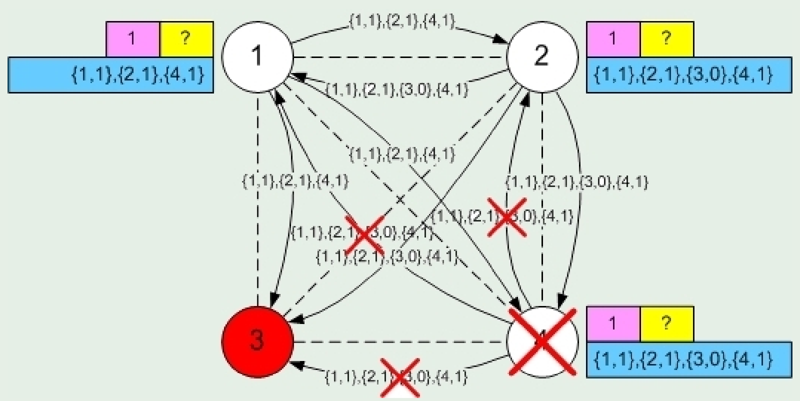
1st Round – process 3 fails



Example of execution of FloodSet algorithm

Let a synchronous complete graph $n = 4$ and $\sigma = 2$.

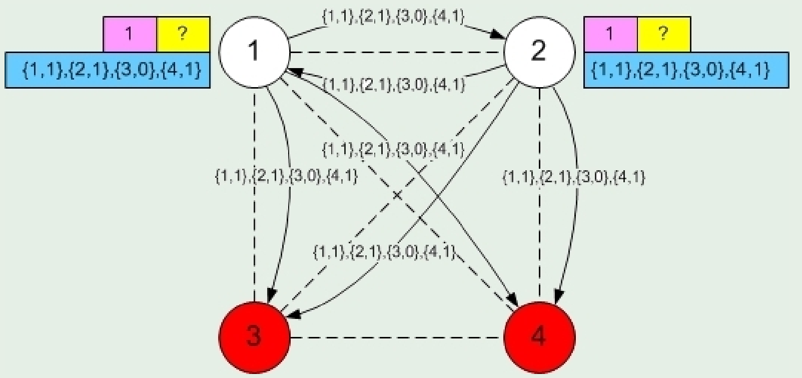
2nd Round – process 4 fails



Example of execution of FloodSet algorithm

Let a synchronous complete graph $n = 4$ and $\sigma = 2$.

3rd Round – no failures

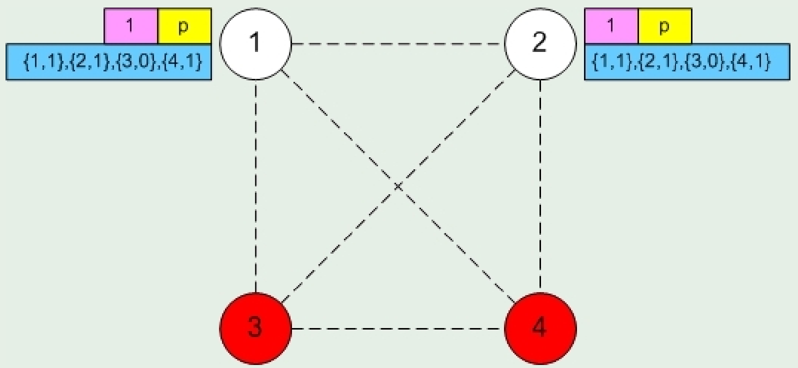


Static Synchronous Networks | Process Failures | Byzantine Failures
An Algorithm for Stopping Failures

Example of execution of FloodSet algorithm

Let a synchronous complete graph $n = 4$ and $\sigma = 2$.

3rd Round – agreement



Ioannis Chatzigiannakis | Pervasive Systems | Lecture 8 34 / 62

Static Synchronous Networks | Process Failures | Byzantine Failures
An Algorithm for Stopping Failures

Properties of FloodSet

Lemma (FloodSet.1)

If no process failes during a particular round $\gamma, 1 \leq \gamma \leq \sigma + 1$, then $I_u(\gamma) = I_v(\gamma)$ for all u and v that are active after γ rounds.

Proof: Suppose that no process fails at round γ and let I be the set of processes that are active after $\gamma - 1$ rounds.
Then, $\forall u \in I$ will send its own $I_u(\gamma)$ to all other processes at the end of round $\gamma - 1$.
Thus at round γ,

$$\forall u \in I, I_u(\gamma) = \cup_{v \in I} I_v(\gamma - 1)$$

■

Static Synchronous Networks | Process Failures | Byzantine Failures
An Algorithm for Stopping Failures

Properties of FloodSet

Lemma (FloodSet.2)

Suppose that $I_u(\gamma) = I_v(\gamma)$ for all u, v that are active after γ rounds. Then for any round $\gamma', \gamma \leq \gamma' \leq \sigma + 1$, the same holds , that is, $I_u(\gamma') = I_v(\gamma')$ for all u, v that are active after γ' rounds.

Proof: All processes that have not failed for γ rounds have identical lists.
The processes that have not failed after γ round still maintain identical lists.
Since no other active process exists, after round γ no new value is circulated in the network.
Therefore the value of $I_u, \forall u \in I$ will not change in any consecutive round.

■

Ioannis Chatzigiannakis | Pervasive Systems | Lecture 8 35 / 62

Properties of FloodSet

Lemma (FloodSet.3)

If processes u, v are both active after $\sigma + 1$ rounds, then $l_u(\sigma + 1) = l_v(\sigma + 1)$ at the end of round $\sigma + 1$.

Proof: Since there are at most σ failures, there must be a round $\gamma, 1 \leq \gamma \leq \sigma + 1$ where no process fails.

- According to lemma FloodSet.1 $l_u(\gamma) = l_v(\gamma)$ for each u, v that are still active after round γ
- According to lemma FloodSet.2 $l_u(\sigma + 1) = l_v(\sigma + 1)$ for each u, v that are still active after round $\sigma + 1$

■

Properties of FloodSet

Theorem

Algorithm FloodSet solves the agreement problem for stopping failures.

Proof:

Termination condition holds – all processes that are active until the end of round $\sigma + 1$, terminate.

Validity condition holds –

- If all processes have initial value τ then the list transmitted is $\{\tau\}$
- The list l_u will not changed at the end of round $\sigma + 1$

Agreement condition holds –

- According to FloodSet.3

■

Properties of FloodSet

- Time complexity is $\sigma + 1$ rounds
- Message complexity is $\mathcal{O}\left((\sigma + 1) \cdot n^2\right)$
- Each message may be of size $\mathcal{O}(n)$ bits
- Communication complexity in bits is $\mathcal{O}\left((\sigma + 1) \cdot n^3\right)$

Alternative rules

- Instead of a predefined value $i_0 \in S$, choose $\min(S)$
- Processes send only messages when they detect a change in their list (OptFloodSet)

The Commit Problem

The processes of the system participate in a transaction. Each process, according to local knowledge decides if the transaction ought to be "committed" or "aborted". If processes wish to "commit" then the outcome should be "commit". If at least one wishes to "abort" then they should all "abort".

- The input domail is {0,1} where 1 represents "commit" and 0 represents "abort".
- In distributed system with multiple databases, in regular time intervals they consolidate their records – depending on the outcome of the consolidation process the servers commit or abort the consolidation transaction.

We assume processes correctly solve the commit problem when the following conditions are satisfied:

1. Agreement: Every pair of processes does not agree on different output values, that is, $\nexists u, v : o_u \neq o_v$
2. Validity:
   - If a process u has input value $i_u =$ "abort" then the only possible output value is "abort".
   - If $\forall u : i_u =$ "commit" and there are no failures then all processes output "commit".
3. Terminate:
   - week – if there are no failures, then all processes eventually decide.
   - strong – all nonfaulty processes eventually decide.

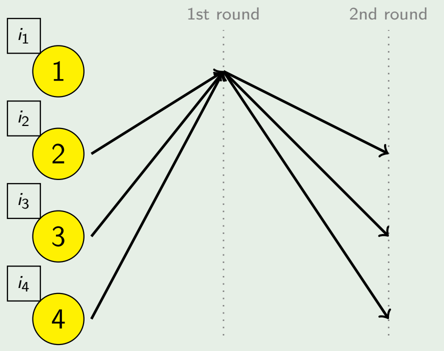
TwoPhaseCommit Algorithm

The algorithm assumes a distinguished process, say u_1.

Round 1 – All processes except u_1 send their initial values to u_1. Process u_1 collects all these value, plus its own initial value, into a vector. If the vector is filled with "commit", then process i_u decides "commit". Otherwise, it decides "abort".

Round 2 – Process u_1 broadcasts its deciision to all other processes. All processes decide on the value received from u_1.

Execution of TwoPhaseCommit – message transmission diagram



Properties of TwoPhaseCommit Algorithm

- The TwoPhaseCommit solves the commit problem under the week termination condition.
- If a stopping failure occurs in u_1 before the end of Round 1 – the algorithm blocks.
- The time complexity is fixed – 2 rounds.
- The communication complexity is $\mathcal{O}(n)$

ThreePhaseCommit Algorithm

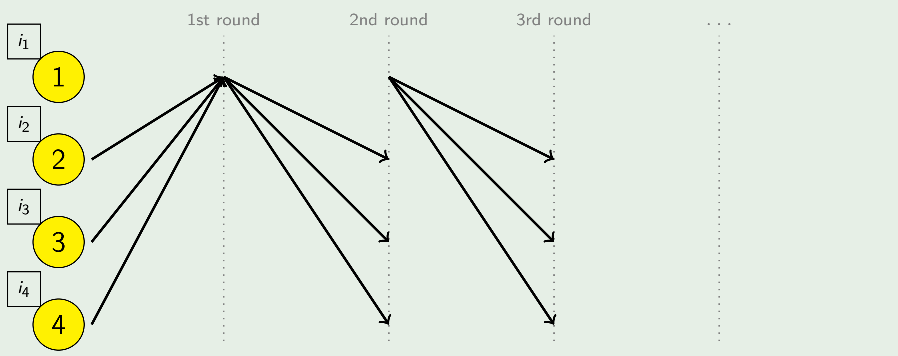
The algorithm assumes the election of a distinguished process, say u_1.

Round 1 – All processes except u_1 send their initial values to u_1. Process 1 collects all these value, plus its own value, into a vector. If all positions are "commit", then u_1 becomes "ready" but does not decide. Otherwise, it decides "abort".

Round 2 – If u_1 decides $o_1 =$ "abort" sends a message "abort", otherwise it sends "ready". Any process that receives "abort" it decides "abort". Any process that receives "ready" becomes "ready". If u_1 is "ready" then it decides $o_1 =$ "commit".

Round 3 – If process u_1 decided "commit" it sends a message "commit" to all other processes. Any process receiving "commit" decides "commit".

Execution of ThreePhaseCommit – message transmission diagram



Any process may end up in one and only one state:

1. κ_0 – decide $o_u =$ "abort"
2. κ_1 – decide $o_u =$ "commit"
3. "ready" – not decided yet, but is "ready" to commit
4. "unknown" – not decided yet and not "ready" to commit

Lemma (ThreePhaseCommit.1)

After three rounds of ThreePhaseCommit algorithm the following are true:

1. *If any process's state is in ready or κ_1, then all processes' initial values are "ready".*
2. *If any process's start is in κ_0, then no process is in κ_1 and no non-failed process is in ready.*
3. *If any process's state is in κ_1, then no process is in κ_0, and no non-failed process is in uncertain.*

Lemma (ThreePhaseCommit.2)

After three rounds of ThreePhaseCommit, the following are true:

1. *The agreement condition holds.*
2. *The validity condition holds.*
3. *If process u_1 has not failed, then all non-failed processes have decided.*

Proof: The agreement condition follows from Lemma ThreePhaseCommit.1.

The agreement condition follows

- partially from Lemma ThreePhaseCommit.1 – if a process starts with "abort", then all decide "abort",
- investigate all other cases.

If process u_1 does not fail, then all active processes have decided.

- Process u_1 decides if no failure occurs.
- Transmits the decision to all other processes.
- All processes that receive the message and do not fail, decide.

■

- The three first rounds are not enough to solve the problem under the strong termination condition.
- If process u_1 fails, some processes may remain in state "unknown".
- The processes execute a "termination protocol".

ThreePhaseCommit Algorithm

Round 4 – All (not yet failed) processes send their current stutus (κ_0, κ_1, "ready", "unknown") to u_2 that puts them in a vector. If the vector:

1. contains at least one κ_0 and u_2 has not yet decided, it decides "abort".
2. contains at least one κ_1 and u_2 has not yet decided, it decides "commit".
3. all values are "unknown" then u_2 decides "abort".
4. all values are either "unknown" or "ready", then u_2 becomes "ready".

ThreePhaseCommit Algorithm

Round 5 – If process u_2 decided to "abort" it sends an "abort" message, while if if decided to "commit" it sends a message "ready". If u_2 has not yet decided it sends "ready". Any process receiving a "abort" or "commit" decides accordingly – if it receives "ready" it becomes "ready". If u_2 has not decided, it decides "commit".

Round 6 – If process u_2 decided "commit" it sends out a message "commit" to all other processes. Any process receiving a "commit" message and has not decided yet, it decides "commit".

The protocol repeats the last three similar rounds coordinated by each process $u \in [3, n]$.

Properties of ThreePhaseCommit algorithm

- The ThreePhaseCommit algorithm solves the commit problem under the strong termination condition
  - By induction on the number of rounds.
  - Based on Lemmas ThreePhaseCommit.1, ThreePhaseCommit.2
- Time complexity is $3n$ rounds – $\mathcal{O}(n)$
- Communication complexity is $\mathcal{O}(n^2)$

Byzantine Failures

- The network includes faulty processes that do not terminate but continue to participate in the execution of the algorithm.
- The behavior of the processes may be completely unpredictable.
- The internal state of a faulty process may change during the execution of a round arbitrarily, without receiving any message.
- A faulty process may send a message with any content (i.e., fake messages), independently of the instructions of the algorithm.
- We call such kind of failures as Byzantine failures.
- We use byzantine failures to model malicious behavior (e.g. cyber-security attacks).

Why study Byzantine Fault Tolerance?

- Does this happen in the real world?
  The "one in a million" case.
  - Malfunctioning hardware,
  - Buggy software,
  - Compromised system due to hackers.
- Assumptions are vulnerabilities.
- Is the cost worth it?
  - Hardware is always getting cheaper,
  - Protocols are getting more and more efficient.

Coordinated Attack of 4 Byzantine Generals

Four generals wish to coordinate the attack of their armies in an enemy city. Among the generals there exits a traitor. All loyal generals must agree to the same attack (or retreat) plan regardless of the actions of the traitor. Communication among generals is carried out by messengers. The traitor is free to do as he chooses.

- Consensus problem in a system with $n = 4$ processes under the presence of byzantine failures.
- Possible input/output values are "yes" or "no" – that is $S = \{"yes", "no"\}$

Problem Statement

- On general achieves the role of Chief of Staff.
- The Chief of Staff has to send an order to each of the $n - 1$ generals such that:
  1. All faithful generals follow the same order
     (all non faulty processes receive the same message)
  2. If the Chief of Staff is faithful, then all faithful generals follow his orders
     (if all processes are non-faulty then the messages received are the same with the transmitting process)
- The above conditions are known as the conditions for "consistent broadcast".
- Note: If the Chief of Staff is faithful, then the 1st condition derives from the 2nd. But he may be the traitor.

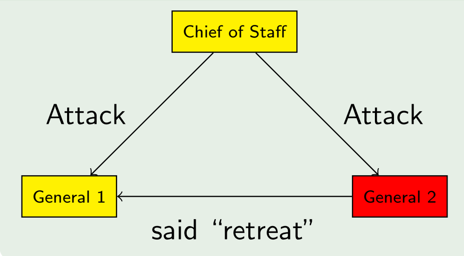
Discussion

- A solution for the Byzantine Generals problems allows:
  1. Reliable communication in the presence of tampered messages
  2. Reliable communication in the presence of message omissions
- Dealing with message omissions (link/stopping failures) is the most common approach.
- We name faults Byzantine all faults that fall under these two categories.
- All solutions to the problem require a network size at least three times the number of failures – that is $n > 3\beta$.
  - Different situation from stopping failures where n and σ did not follow any relationship.
  - May sound surprising high, due to the *triple-modular redundancy* – that states that $n > 2\beta + 1$.

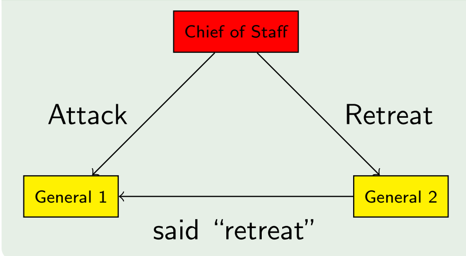
Impossibility result

Let's examine the following cases involving 3 generals:

Case #1



Case #2



- In case #1, General 1 in order to meet the 2nd condition, he has to attack.

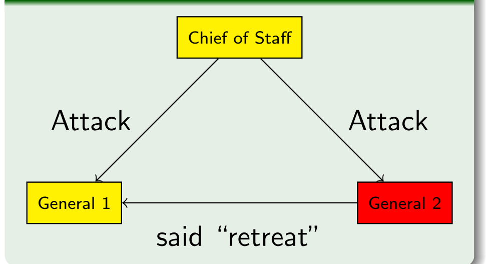
2nd Condition

If the Chief of Staff is faithful, then all faithful generals follow his orders.

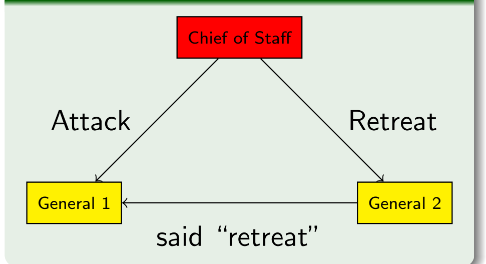
Impossibility result

Let's examine the following cases involving 3 generals:

Case #1



Case #2



- In case #2, if General 1 attacks then he violates the 1st condition.

1st Condition

All faithful generals follow the same order

Impossibility result

- Given the messages received by General 1, each case looks symmetric.
- General 1 cannot break the symmetry.
- No solution exists for the Byzantine Generals in case of 3 generals and 1 traitor.
- Generalization of the impossibility result:
  No solution exists for less then $3\beta + 1$ generals if it has to deal with β traitors.

Lamport, Shostak and Pease Algorithm

L. Lamport, R. Shostak, M. Pease: "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, 4(3): pp 382-401, 1982.

- The algorithm makes three assumptions regarding communication:
  1. All message transmissions are delivered correctly.
  2. The receivers knows the identity of the sender.
  3. The absence of a message can be detected.
- The 1st and 2nd assumptions limit the traitor from interfering with the transmissions of the other generals.
- The 3rd assumptions prevents the traitor to delay the attack by not sending any message.
- In computer networks conditions 1 and 2 assume that the processors are directly connected and communication failures are counted as part of the β failures.

Lamport, Shostak and Pease Algorithm

- Let n processes and β failures.
- Processes have a predefined decision o_{def} that is used when the Chief of Staff is a traitor (e.g., retreat).
- We define function majority$(o_1, \ldots, o_{n-1}) = o$ that computes the majority of decisions $o_u = o$

Algorithm UM(n,0) (for 0 traitors)

1. The Chief of Staff transmits decision o to all generals.
2. All generals decide o or if they do not receive a message, they decide o_{def}.

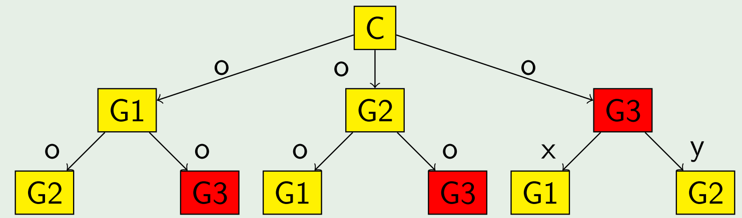
Lamport, Shostak and Pease Algorithm

Algorithm UM(n, m) (for m traitors)

1. The Chief of Staff transmits decision o to all generals.
2. For each general u
   - Set o_u to the value received, or if no message received, set to o_{def}.
   - Send the value o_u to the $n-2$ generals by invoking UM$(n-1, m-1)$.
3. For each general u and each $v \neq u$
   - Set o_v to the value received from u at step 2, or if no message received set to o_{def}.
   - Decide on value majority$(o_1, \ldots, o_{n-1})$.

Example of Execution

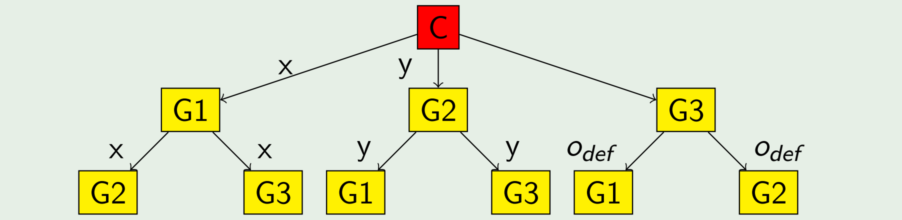
$n = 4, \beta = 1$ – G3 is the traitor



- At the end of 1st phase: G1 ($o_1 = o$), G2 ($o_2 = o$), G3 ($o_3 = o$)
- At the end of 2nd phase:
  - G1 – $o_1 = o, o_2 = o, o_3 = x$
  - G2 – $o_1 = o, o_2 = o, o_3 = y$
  - G3 – $o_1 = o, o_2 = o, o_3 = o$
- At the end of 2nd phase, each general has the same number of values and reaches the same decision due to condition 1.
- The decision of the Chief coincides with the majority (2nd condition)

Example of Execution

$n = 4, \beta = 1$ – the Chief of Staff is the traitor



- At the end of 1st phase: G1 ($o_1 = x$), G2 ($o_2 = y$), G3 ($o_3 = o_{def}$)
- At the end of 2nd phase:

  G1 – $o_1 = x, o_2 = y, o_3 = o_{def}$
  G2 – $o_1 = x, o_2 = y, o_3 = o_{def}$
  G3 – $o_1 = x, o_2 = y, o_3 = o_{def}$
- The three loyal generals decide majority(x, y, o_{def}) thus both 1st and 2nd conditions are met.

Lemma

For any m and k, UM(m) adheres the 2nd condition given $2k + m$ generals and at most k traitors.

Proof: (By induction on m)
In the 1st step, UM(0) works if the Chief of Staff is loyal, i.e. UM(0) meets the 2nd condition.
Let's assume that UM($m - 1$) meets the 2nd condition for $m > 0$.
We can show that it holds for m:

- In the 1st step, the loyal general sends the value o to $n - 1$ generals.
- In the 2nd step all loyal general execute UM($m - 1$).
- From the original assumption it holds that $n > 2k + m$ or $n - 1 > 2k + (m - 1)$.

- From the induction step that we defined, each loyal general u receives $o_u = o_v$ from each loyal general v.
- Since there are at most k traitors and $n - 1 > 2k + (m - 1) \geq 2k$, i.e., $k < \frac{n-1}{2}$ then the majority is reached from the $n - 1$ loyal generals.
- Thus each loyal general has $o_u = o$ for majority of $n - 1$ values – thus in the 3rd step, by invoking majority($o_1, \ldots, o_{n-1}$) it outputs o that meets the 2nd condition.

■

Theorem

For any m, UM(m) adheres the 1st and 2nd condition given $3m$ generals and at most m traitors.

Proof: (By induction on m)
If no traitors exists it is easy to show that with the user of the algorithm, in the 1st step, conditions 1 and 2 hold.
If we assume that UM($m - 1$) meets conditions 1 and 2 for $m > 0$.
We can show for m:
Case 1:

- Assume the Chief of Staff is loyal.
- For $k = m$ due to the Lemma, UM(m) meets the 2nd condition.
- Since the 1st condition derives from the 2nd condition when the Chief of Staff is loyal, it is enough to show the second case:

Case 2:

- The Chief of Staff is a traitor.
- There exist at most m traitors and the Chief of Staff is among them.
- Thus, at most $m-1$ generals are traitors.
- Since we have $3m$ generals, the loyal generals must be $3m-1 > 3(m-1)$
- Therefore we can apply the inductive step and conclude that UM$(m-1)$ meets the 1st and 2nd condition.
- Thus for each v, each pair of loyal generals receives the same value o_v in the 3rd step.
- Thus, each pair of loyal generals receives the same number of values and thus majority$(o_1, \ldots, o_{n-1})$ returns the same value – which meets the 1st condition.

■

Properties of Algorithm

- By applying UM(n,β) we get $n-1$ messages
- For each message the UM(n,$\beta-1$) is activated that generates $n-2$ messages
- ...
- The total number of messages is $\mathcal{O}(n^{\beta+1})$
- The $\beta+1$ steps during which messages are exchanged between the processes is a mandatory feature of algorithms that need to reach consensus in the presence of β faulty processes.