Why are distributed systems so complicated and what can we do about it?

Maarten van Steen
m.r.vansteen@utwente.nl
`https://www.distributed-systems.net`

Version: August 16, 2018

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

About this tutorial

About this tutorial

Shocking?

I don’t want to teach you anything specifically ...

About this tutorial

Shocking?

I don't want to teach you anything specifically ...

Ultimate goal

... except that I want to propagate simple designs, where possible.

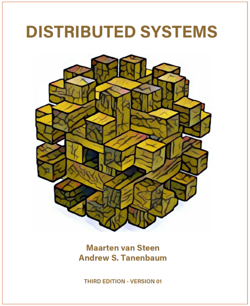
About this tutorial

Shocking?

I don’t want to teach you anything specifically ...

Ultimate goal

... except that I want to propagate simple designs, where possible.



DISTRIBUTED SYSTEMS

Maarten van Steen
Andrew S. Tanenbaum

THIRD EDITION - VERSION 01

https://www.distributed-systems.net

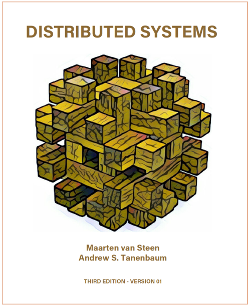
About this tutorial

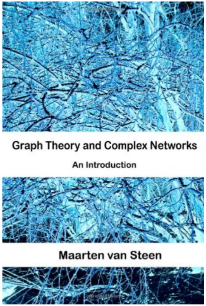
Shocking?

I don’t want to teach you anything specifically ...

Ultimate goal

... except that I want to propagate simple designs, where possible.

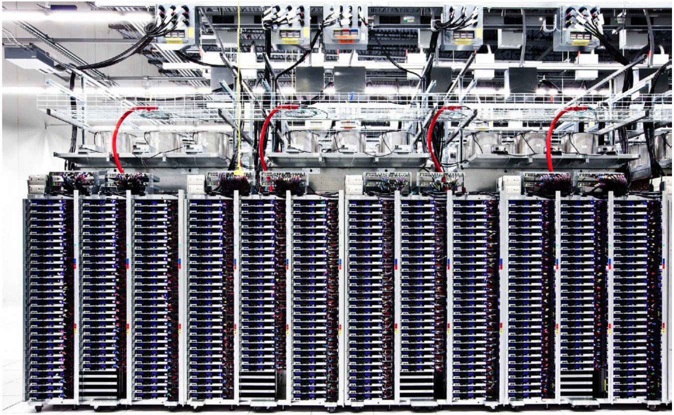




https://www.distributed-systems.net

Introduction and overview

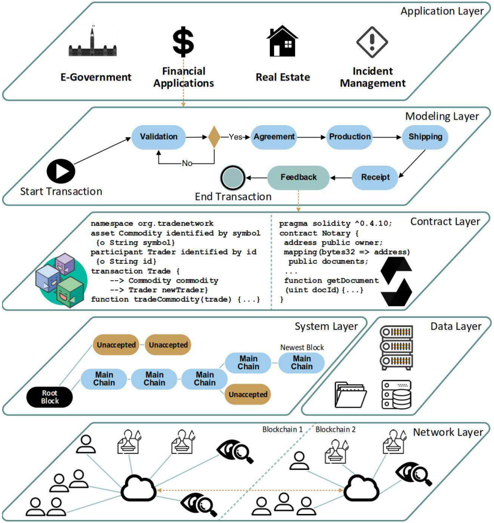
Modern distributed systems: Google



“*Our hardware must be controlled and administered by software that can handle massive scale.*”

Site Reliability Engineering, O’Reilly, 2016

Modern distributed systems: Blockchain

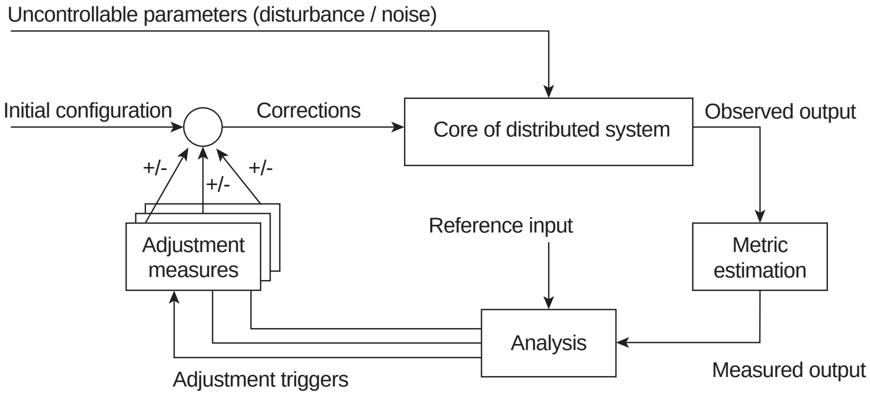


```
namespace org.tradenetwork
asset Commodity identified by symbol
 {o String symbol}
participant Trader identified by id
 {o String id}
transaction Trade {
    --> Commodity commodity
    --> Trader newTrader}
function tradeCommodity(trade) {...}
```

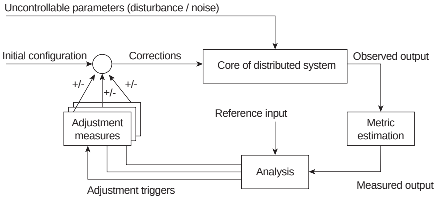
```
pragma solidity ^0.4.10;
contract Notary {
 address public owner;
 mapping(bytes32 => address)
  public documents;
 ...
 function getDocument
 (uint docId){...}
}
```

Zang, Vitenberg, Jacobsen, “Deconstructing Blockchains”, 2018

Distributed systems as feedback control loops



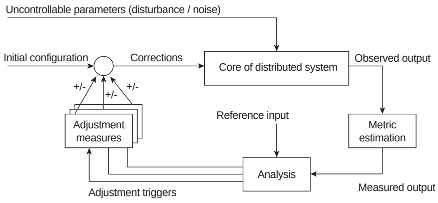
Distributed systems as feedback control loops



Handling faults

- We observe (lack of) responses
- We use timeouts as our metric of choice
- We measure response times (or timeouts)
- Our reference input consists of accepted timings
- Analytics shows that response times are too high
- We conclude that a primary server is not responsive
- Backups are “instructed” to enter special state

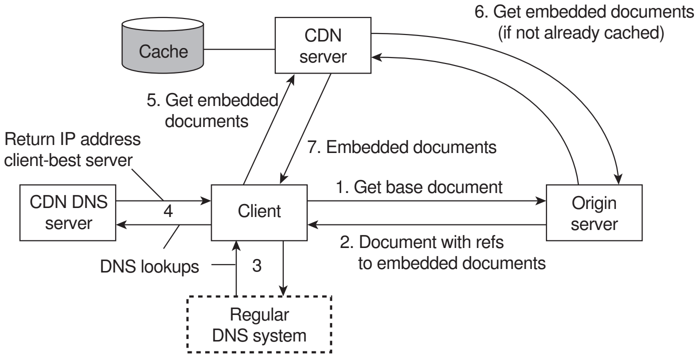
Distributed systems as feedback control loops



Dynamic replication in CDNs

- We observe access patterns for Web content
- We use geographic distance between client and edge server as metric
- We estimate location of the client
- Analytics shows that client is best served by a specific edge server
- We select that edge server to host the Web content
- Client is directed to edge server, which subsequently loads the content into its cache

Dynamic replication in a CDN



Three questions

1. What can make a distributed system so complicated?
2. Are there ways to keep matters simple?
3. To what extent can we ignore details?

Three questions

1. What can make a distributed system so complicated?

   Focus on distributed consensus

2. Are there ways to keep matters simple?

3. To what extent can we ignore details?

Three questions

1. What can make a distributed system so complicated?

   Focus on distributed consensus

2. Are there ways to keep matters simple?

   Focus on gossiping for replication

3. To what extent can we ignore details?

Three questions

1. What can make a distributed system so complicated?

   Focus on distributed consensus

2. Are there ways to keep matters simple?

   Focus on gossiping for replication

3. To what extent can we ignore details?

   Focus on cognification for predictive maintenance

Distributed consensus protocols

Dependability

Basics

A component provides services to clients. To provide services, the component may require the services from other components ⇒ a component may depend on some other component.

Specifically

A component C depends on C^* if the correctness of C's behavior depends on the correctness of C^*'s behavior. (Components are processes or channels.)

Dependability

Basics

A component provides services to clients. To provide services, the component may require the services from other components ⇒ a component may depend on some other component.

Specifically

A component C depends on C^* if the correctness of C's behavior depends on the correctness of C^*'s behavior. (Components are processes or channels.)

Requirements related to dependability

Requirement	**Description**
Reliability	Continuity of service delivery
Availability	Readiness for usage
Safety	Very low probability of catastrophes
Maintainability	How easy can a failed system be repaired

Reliability versus availability

Reliability $R(t)$ of component C

Conditional probability that C has been functioning correctly during $[0,t)$ given C was functioning correctly at time $T=0$.

Traditional metrics

- Mean Time To Failure (*MTTF*): The average time until a component fails.
- Mean Time To Repair (*MTTR*): The average time needed to repair a component.
- Mean Time Between Failures (*MTBF*): Simply $MTTF + MTTR$.

Reliability versus availability

Availability $A(t)$ of component C

Average fraction of time that C has been up-and-running in interval $[0,t)$.

- Long-term availability A: $A(\infty)$
- Note: $A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF+MTTR}$

Observation

Reliability and availability make sense only if we have an accurate notion of what a failure actually is.

Terminology

Failure, error, fault

Term	Description	Example
Failure	A component is not living up to its specifications	Crashed program
Error	Part of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

Terminology

Handling faults

Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don’t hire sloppy programmers
Fault tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

Dependability versus security

Omission versus commission

Arbitrary failures are sometimes qualified as malicious. It is better to make the following distinction:

- Omission failures: a component fails to take an action that it should have taken
- Commission failure: a component takes an action that it should not have taken

Dependability versus security

Omission versus commission

Arbitrary failures are sometimes qualified as malicious. It is better to make the following distinction:

- Omission failures: a component fails to take an action that it should have taken
- Commission failure: a component takes an action that it should not have taken

Observation

Note that deliberate failures, be they omission or commission failures are typically security problems. Distinguishing between deliberate failures and unintentional ones is, in general, impossible.

Redundancy for failure masking

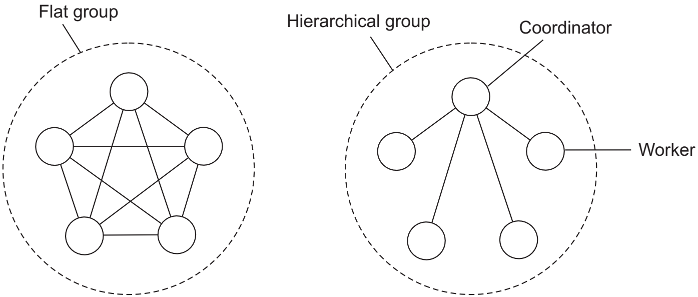
Types of redundancy

- Information redundancy: Add extra bits to data units so that errors can be recovered when bits are garbled.
- Time redundancy: Design a system such that an action can be performed again if anything went wrong. Typically used when faults are transient or intermittent.
- Physical redundancy: add equipment or processes in order to allow one or more components to fail. This type is extensively used in distributed systems.

Process resilience

Basic idea

Protect against malfunctioning processes through process replication, organizing multiple processes into a process group. Distinguish between flat groups and hierarchical groups.



Groups and failure masking

k-fault tolerant group

When a group can mask any *k* concurrent member failures (*k* is called degree of fault tolerance).

Groups and failure masking

k-fault tolerant group

When a group can mask any k concurrent member failures (k is called degree of fault tolerance).

How large does a *k*-fault tolerant group need to be?

- With halting failures: we need a total of $k+1$ members as no member will produce an incorrect result, so the result of one member is good enough.
- With arbitrary failures: we need $2k+1$ members so that the correct result can be obtained through a majority vote.

Groups and failure masking

k-fault tolerant group

When a group can mask any k concurrent member failures (k is called degree of fault tolerance).

How large does a *k*-fault tolerant group need to be?

- With halting failures: we need a total of $k+1$ members as no member will produce an incorrect result, so the result of one member is good enough.
- With arbitrary failures: we need $2k+1$ members so that the correct result can be obtained through a majority vote.

Important assumptions

- All members are identical
- All members process commands in the same order (this assumption complicates matters)

Result: We can now be sure that all processes do exactly the same thing.

Consensus

Prerequisite

In a fault-tolerant process group, each nonfaulty process executes the same commands, and in the same order, as every other nonfaulty process.

Reformulation

Nonfaulty group members need to reach consensus on which command to execute next.

Flooding-based consensus

System model

- A process group $\mathbf{P} = \{P_1, \ldots, P_n\}$
- Fail-stop failure semantics, i.e., with reliable failure detection
- A client contacts a P_i requesting it to execute a command
- Every P_i maintains a list of proposed commands

Flooding-based consensus

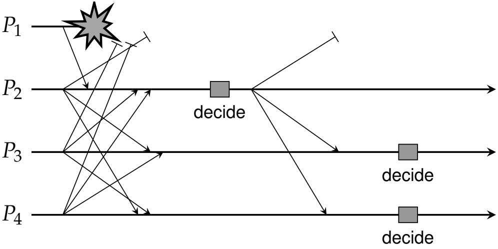
System model

- A process group $\mathbf{P} = \{P_1, \ldots, P_n\}$
- Fail-stop failure semantics, i.e., with reliable failure detection
- A client contacts a P_i requesting it to execute a command
- Every P_i maintains a list of proposed commands

Basic algorithm (based on rounds)

1. In round r, P_i multicasts its known set of commands $\mathbf{C_i^r}$ to all others
2. At the end of r, each P_i merges all received commands into a new $\mathbf{C_i^{r+1}}$.
3. Next command cmd_i selected through a globally shared, deterministic function: $cmd_i \leftarrow select(\mathbf{C_i^{r+1}})$.

Flooding-based consensus: Example



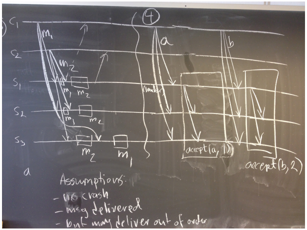
Observations

- P_2 received all proposed commands from all other processes ⇒ makes decision.
- P_3 may have detected that P_1 crashed, but does not know if P_2 received anything, i.e., P_3 cannot know if it has the same information as P_2 ⇒ cannot make decision (same for P_4).

Understanding Paxos

Largely based on joint work with

Hein Meling
(Univ. Stavanger)



Assumptions:
- no crash
- msg delivered
- but may deliver out of order

Written up in
H. Meling, L. Jehl: “Paxos Explained from Scratch.” In Proc. 17th Int’l Conf. Principles of Distributed Systems (OPODIS), Lecture Notes in Computer Science, vol. 8304, pp. 1–10. Springer, 2013.

Realistic consensus: Paxos

Assumptions (rather weak ones, and realistic)

- A partially synchronous system (in fact, it may even be asynchronous).
- Communication between processes may be unreliable: messages may be lost, duplicated, or reordered.
- Corrupted message can be detected (and thus subsequently ignored).
- All operations are deterministic: once an execution is started, it is known exactly what it will do.
- Processes may exhibit halting failures, but not arbitrary failures (note: makes Paxos not suitable for some blockchains).

Understanding Paxos

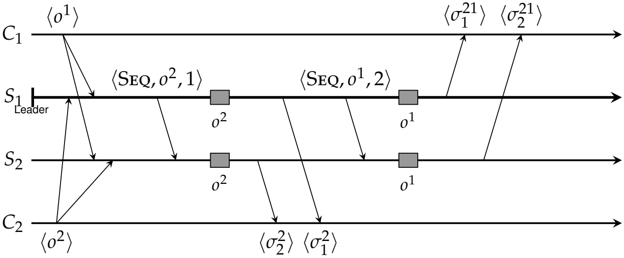
We will build up Paxos from scratch to understand where many consensus algorithms actually come from.

Paxos essentials

Starting point

- We assume a client-server configuration, with initially one primary server.
- To make the server more robust, we start with adding a backup server.
- To ensure that all commands are executed in the same order at both servers, the primary assigns unique sequence numbers to all commands. In Paxos, the primary is called the leader.
- Assume that actual commands can always be restored (either from clients or servers) ⇒ we consider only control messages.

Two-server situation

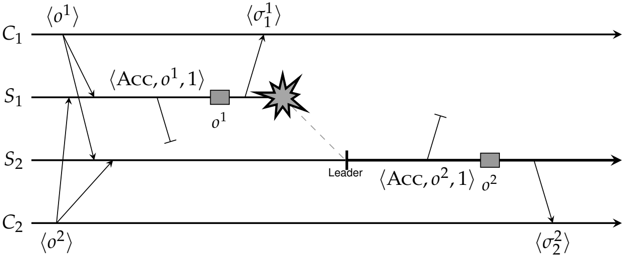

C_1
S_1
Leader
S_2
C_2
$\langle o^1 \rangle$
$\langle o^2 \rangle$
$\langle \text{SEQ}, o^2, 1 \rangle$
$\langle \text{SEQ}, o^1, 2 \rangle$
o^2
o^2
o^1
o^1
$\langle \sigma_1^{21} \rangle$
$\langle \sigma_2^{21} \rangle$
$\langle \sigma_2^{2} \rangle$
$\langle \sigma_1^{2} \rangle$

Handling lost messages

Some Paxos terminology

- The leader sends an accept message $\text{ACCEPT}(o, t)$ to backups when assigning a timestamp t to command o.
- A backup responds by sending a learn message: $\text{LEARN}(o, t)$
- When the leader notices that operation o has not yet been learned, it retransmits $\text{ACCEPT}(o, t)$ with the original timestamp.

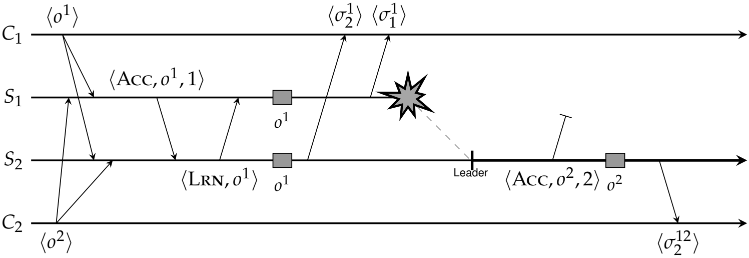
Two servers and one crash: problem


C_1
S_1
S_2
C_2
$\langle o^1 \rangle$
$\langle \sigma_1^1 \rangle$
$\langle \text{Acc}, o^1, 1 \rangle$
o^1
Leader
$\langle \text{Acc}, o^2, 1 \rangle$
o^2
$\langle o^2 \rangle$
$\langle \sigma_2^2 \rangle$

Problem

Primary crashes after executing an operation, but the backup never received the accept message.

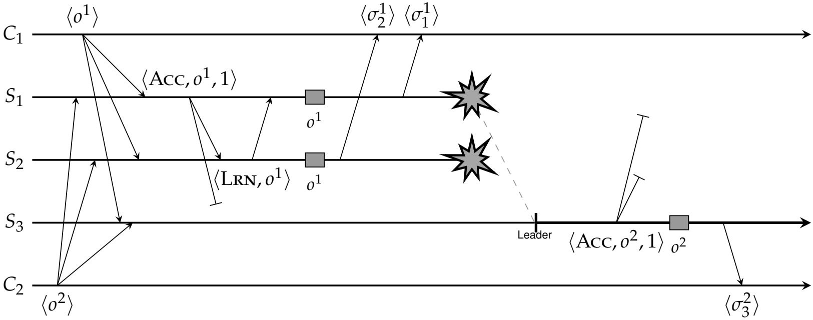
Two servers and one crash: solution


C_1
S_1
S_2
C_2
$\langle o^1 \rangle$
$\langle \sigma_2^1 \rangle$
$\langle \sigma_1^1 \rangle$
$\langle \text{Acc}, o^1, 1 \rangle$
o^1
$\langle \text{Lrn}, o^1 \rangle$
o^1
Leader
$\langle \text{Acc}, o^2, 2 \rangle$
o^2
$\langle o^2 \rangle$
$\langle \sigma_2^{12} \rangle$

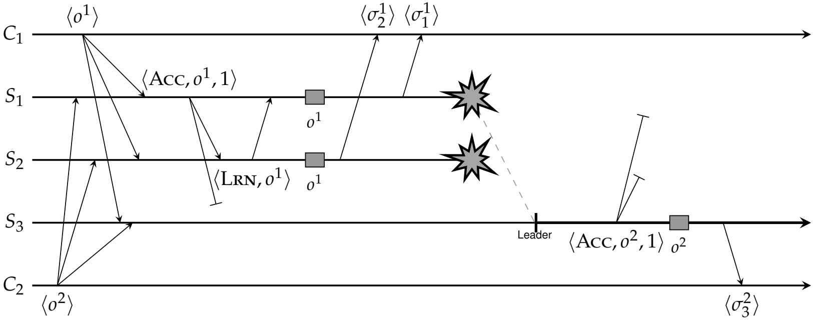
Solution

Never execute an operation before it is clear that is has been learned.

Three servers and two crashes: still a problem?


$\langle o^1 \rangle$
$\langle \sigma_2^1 \rangle$
$\langle \sigma_1^1 \rangle$
C_1
$\langle \textsc{Acc}, o^1, 1 \rangle$
S_1
o^1
S_2
$\langle \textsc{Lrn}, o^1 \rangle$
o^1
S_3
Leader
$\langle \textsc{Acc}, o^2, 1 \rangle$
o^2
C_2
$\langle o^2 \rangle$
$\langle \sigma_3^2 \rangle$

Three servers and two crashes: still a problem?

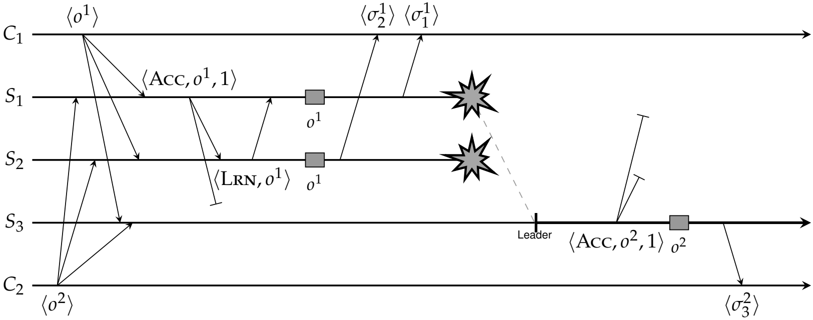


$\langle o^1 \rangle$
$\langle \sigma^1_2 \rangle$
$\langle \sigma^1_1 \rangle$
C_1
$\langle \textsc{Acc}, o^1, 1 \rangle$
S_1
o^1
S_2
$\langle \textsc{Lrn}, o^1 \rangle$
o^1
S_3
Leader
$\langle \textsc{Acc}, o^2, 1 \rangle$
o^2
C_2
$\langle o^2 \rangle$
$\langle \sigma^2_3 \rangle$

Scenario

What happens when LEARN(o^1) as sent by S_2 to S_1 is lost?

Three servers and two crashes: still a problem?


C_1
S_1
S_2
S_3
C_2
$\langle o^1 \rangle$
$\langle \sigma_2^1 \rangle$
$\langle \sigma_1^1 \rangle$
$\langle \text{ACC}, o^1, 1 \rangle$
o^1
$\langle \text{LRN}, o^1 \rangle$
o^1
Leader
$\langle \text{ACC}, o^2, 1 \rangle$
o^2
$\langle o^2 \rangle$
$\langle \sigma_3^2 \rangle$

Scenario

What happens when LEARN(o^1) as sent by S_2 to S_1 is lost?

Solution

S_2 will also have to wait until it knows that S_3 has learned o^1.

Paxos: fundamental rule

General rule

In Paxos, a server *S* cannot execute an operation *o* until it has received a LEARN(*o*) from all other nonfaulty servers.

Failure detection

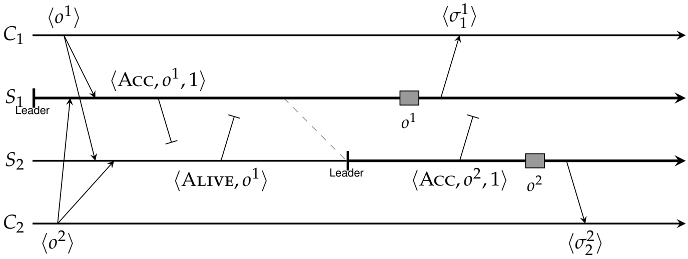
Practice

Reliable failure detection is practically impossible. A solution is to set timeouts, but take into account that a detected failure may be false.

Failure detection

Practice

Reliable failure detection is practically impossible. A solution is to set timeouts, but take into account that a detected failure may be false.


C_1
$\langle o^1 \rangle$
$\langle \sigma_1^1 \rangle$
S_1
Leader
$\langle \text{ACC}, o^1, 1 \rangle$
o^1
S_2
$\langle \text{ALIVE}, o^1 \rangle$
Leader
$\langle \text{ACC}, o^2, 1 \rangle$
o^2
C_2
$\langle o^2 \rangle$
$\langle \sigma_2^2 \rangle$

Required number of servers

Observation

Paxos needs at least three servers

Required number of servers

Observation

Paxos needs at least three servers

Adapted fundamental rule

In Paxos with three servers, a server *S* cannot execute an operation *o* until it has received at least one (other) LEARN(*o*) message, so that it knows that a majority of servers will execute *o*.

Required number of servers

Assumptions before taking the next steps

- Initially, S_1 is the leader.
- A server can reliably detect it has missed a message, and recover from that miss.
- When a new leader needs to be elected, the remaining servers follow a strictly deterministic algorithm, such as $S_1 \rightarrow S_2 \rightarrow S_3$.
- A client cannot be asked to help the servers to resolve a situation.

Required number of servers

Assumptions before taking the next steps

- Initially, S_1 is the leader.
- A server can reliably detect it has missed a message, and recover from that miss.
- When a new leader needs to be elected, the remaining servers follow a strictly deterministic algorithm, such as $S_1 \rightarrow S_2 \rightarrow S_3$.
- A client cannot be asked to help the servers to resolve a situation.

Observation

If either one of the backups (S_2 or S_3) crashes, Paxos will behave correctly: operations at nonfaulty servers are executed in the same order.

Leader crashes after executing o^1

Leader crashes after executing o^1

S_3 is completely ignorant of any activity by S_1

- S_2 received $\text{ACCEPT}(o,1)$, detects crash, and becomes leader.
- S_3 even never received $\text{ACCEPT}(o,1)$.
- S_2 sends $\text{ACCEPT}(o^2,2) \Rightarrow S_3$ sees unexpected timestamp and tells S_2 that it missed o^1.
- S_2 retransmits $\text{ACCEPT}(o^1,1)$, allowing S_3 to catch up.

Leader crashes after executing o^1

S_3 is completely ignorant of any activity by S_1

- S_2 received ACCEPT$(o, 1)$, detects crash, and becomes leader.
- S_3 even never received ACCEPT$(o, 1)$.
- S_2 sends ACCEPT$(o^2, 2) \Rightarrow S_3$ sees unexpected timestamp and tells S_2 that it missed o^1.
- S_2 retransmits ACCEPT$(o^1, 1)$, allowing S_3 to catch up.

S_2 missed ACCEPT$(o^1, 1)$

- S_2 did detect crash and became new leader
- S_2 sends ACCEPT$(o^1, 1) \Rightarrow S_3$ retransmits LEARN(o^1).
- S_2 sends ACCEPT$(o^2, 1) \Rightarrow S_3$ tells S_2 that it apparently missed ACCEPT$(o^1, 1)$ from S_1, so that S_2 can catch up.

Leader crashes after sending ACCEPT(o^1, 1)

S_3 is completely ignorant of any activity by S_1

As soon as S_2 announces that o^2 is to be accepted, S_3 will notice that it missed an operation and can ask S_2 to help recover.

S_2 had missed ACCEPT(o^1, 1)

As soon as S_2 proposes an operation, it will be using a stale timestamp, allowing S_3 to tell S_2 that it missed operation o^1.

Leader crashes after sending ACCEPT(o^1, 1)

S_3 is completely ignorant of any activity by S_1

As soon as S_2 announces that o^2 is to be accepted, S_3 will notice that it missed an operation and can ask S_2 to help recover.

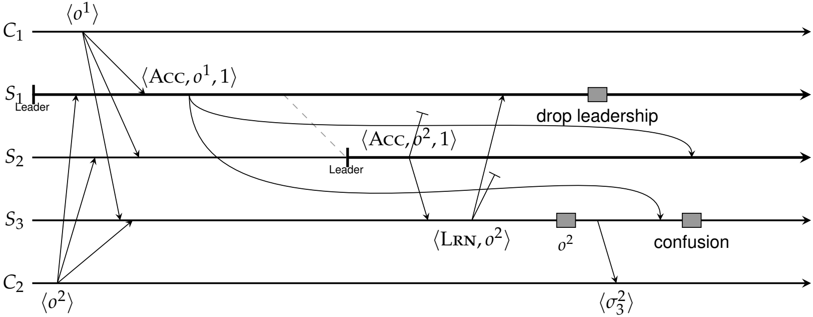
S_2 had missed ACCEPT(o^1, 1)

As soon as S_2 proposes an operation, it will be using a stale timestamp, allowing S_3 to tell S_2 that it missed operation o^1.

Observation

Paxos (with three servers) behaves correctly when a single server crashes, regardless when that crash took place.

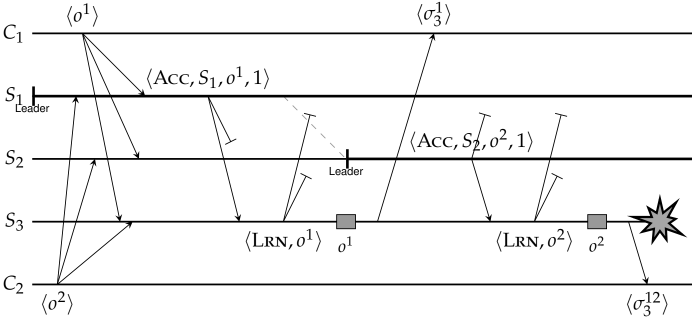
False crash detections


$\langle o^1 \rangle$
C_1
$\langle \textsc{Acc}, o^1, 1 \rangle$
S_1
Leader
drop leadership
$\langle \textsc{Acc}, o^2, 1 \rangle$
S_2
Leader
S_3
$\langle \textsc{Lrn}, o^2 \rangle$
o^2
confusion
C_2
$\langle o^2 \rangle$
$\langle \sigma_3^2 \rangle$

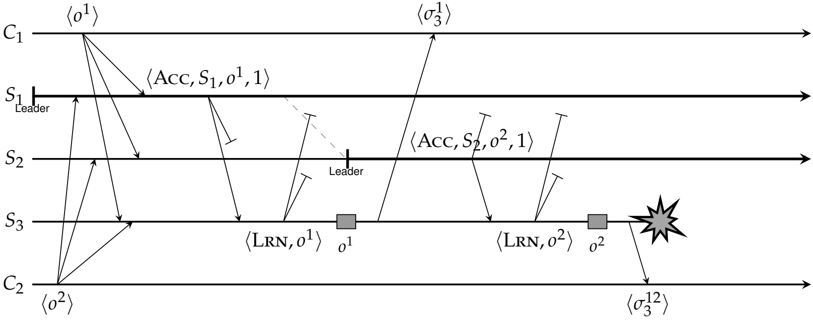
Problem and solution

S_3 receives ACCEPT$(o^1, 1)$, but much later than ACCEPT$(o^2, 1)$. If it knew who the current leader was, it could safely reject the delayed accept message ⇒ leaders should include their ID in messages.

But what about progress?


$\langle o^1 \rangle$
$\langle \sigma_3^1 \rangle$
C_1
$\langle \textsc{Acc}, S_1, o^1, 1 \rangle$
S_1
Leader
$\langle \textsc{Acc}, S_2, o^2, 1 \rangle$
S_2
Leader
S_3
$\langle \textsc{Lrn}, o^1 \rangle$
o^1
$\langle \textsc{Lrn}, o^2 \rangle$
o^2
C_2
$\langle o^2 \rangle$
$\langle \sigma_3^{12} \rangle$

But what about progress?



C_1
S_1
Leader
S_2
Leader
S_3
C_2
$\langle o^1 \rangle$
$\langle o^2 \rangle$
$\langle \text{ACC}, S_1, o^1, 1 \rangle$
$\langle \text{ACC}, S_2, o^2, 1 \rangle$
$\langle \text{LRN}, o^1 \rangle$
$\langle \text{LRN}, o^2 \rangle$
o^1
o^2
$\langle \sigma_3^1 \rangle$
$\langle \sigma_3^{12} \rangle$

Essence of solution

When S_2 takes over, it needs to make sure than any outstanding operations initiated by S_1 have been properly flushed, i.e., executed by enough servers. This requires an explicit leadership takeover by which other servers are informed before sending out new accept messages.

Replication for scalability

Performance and scalability

Main issue

To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the the same order everywhere

Conflicting operations: From the world of transactions

- Read–write conflict: a read operation and a write operation act concurrently
- Write–write conflict: two concurrent write operations

Issue

Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability Solution: weaken consistency requirements so that hopefully global synchronization can be avoided

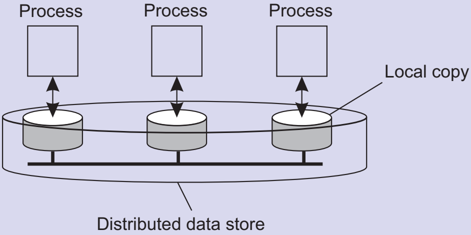
Data-centric consistency models

Consistency model

A contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

Essential

A data store is a distributed collection of storages:

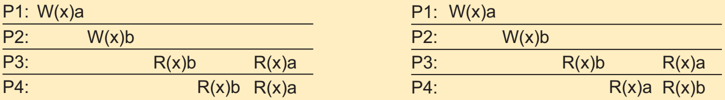


Sequential consistency

Definition

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

(a) A sequentially consistent data store. (b) A data store that is not sequentially consistent



P1: W(x)a
P2: W(x)b
P3: R(x)b R(x)a
P4: R(x)b R(x)a

(a)

P1: W(x)a
P2: W(x)b
P3: R(x)b R(x)a
P4: R(x)a R(x)b

(b)

Grouping operations

Definition

- Accesses to locks are sequentially consistent.
- No access to a lock is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to locks have been performed.

Grouping operations

Definition

- Accesses to locks are sequentially consistent.
- No access to a lock is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to locks have been performed.

Basic idea

You don't care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series itself to be known.

Grouping operations

A valid event sequence for coarse-grained consistency

P1:	L(x)	W(x)a	L(y)	W(y)b	U(x)	U(y)				
P2:						L(x)	R(x)a			R(y) NIL
P3:								L(y)	R(y)b	

Observation

Coarse-grained consistency implies that we need to lock and unlock data (implicitly or not).

Consistency, availability, and partitioning

CAP theorem

Any networked system providing shared data can provide only two of the following three properties:

C: consistency, by which a shared and replicated data item appears as a single, up-to-date copy

A: availability, by which updates will always be eventually executed

P: Tolerant to the partitioning of a process group.

Conclusion

In a network subject to communication failures, it is impossible to realize an atomic read/write distributed shared memory that guarantees a response to every request.

CAP in practice

Understanding CAP

Consider two processes that can no longer communicate:

- Letting one process accept updates leads to inconsistency: $\{A, P\}$.
- If consistency needs to be maintained, one process will have to fake to be unavailable: $\{C, P\}$.
- If we do insist on communication, then we cannot tolerate partitions: $\{C, A\}$.

Practice

Tolerate partitions and let processes simply go ahead accepting potential inconsistencies. Design the system to mitigate the effects of inconsistencies (which can often be done by considering applications).

Eventual consistency

Observation

In practice, we often see that data is read by many and updated by few, essentially enabling reducing the complexity caused by write-write conflicts: only few processes need to ensure that their updates take place in the same order.

Basic idea

Let updates gradually propagate to (read only) replicas. If a client always accesses the same replica, it may not even notice that what it reads is not yet up-to-date. It may also not matter.

Eventual consistency

Observation

In practice, we often see that data is read by many and updated by few, essentially enabling reducing the complexity caused by write-write conflicts: only few processes need to ensure that their updates take place in the same order.

Basic idea

Let updates gradually propagate to (read only) replicas. If a client always accesses the same replica, it may not even notice that what it reads is not yet up-to-date. It may also not matter.

Observation

We need to come up with efficient and effective means for spreading updates across replicas, no matter where they are located in a wide-area distributed system.

Flooding

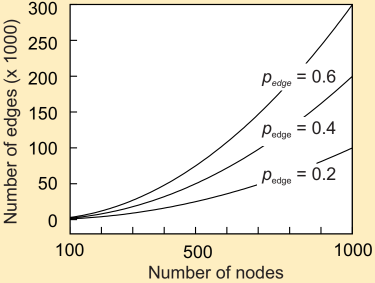
Essence

P simply sends a message *m* to each of its neighbors. Each neighbor will forward that message, except to *P*, and only if it had not seen *m* before.

Performance

The more edges, the more expensive!

The size of a random overlay as function of the number of nodes



Flooding

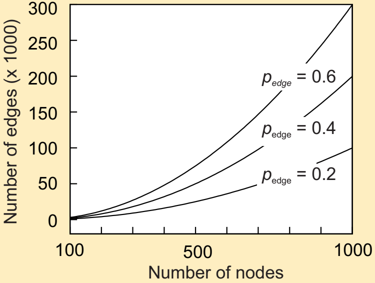
Essence

P simply sends a message *m* to each of its neighbors. Each neighbor will forward that message, except to *P*, and only if it had not seen *m* before.

Performance

The more edges, the more expensive!

The size of a random overlay as function of the number of nodes



Variation

Let *Q* forward a message with a certain probability p_{flood}, possibly even dependent on its own number of neighbors (i.e., node degree) or the degree of its neighbors.

Epidemic protocols

Assume there are no write–write conflicts

- Update operations are performed at a single server
- A replica passes updated state to only a few neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every replica

Two forms of epidemics

- Anti-entropy: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards
- Rumor spreading: A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).

Anti-entropy

Principal operations

- A node P selects another node Q from the system at random.
- Pull: P only pulls in new updates from Q
- Push: P only pushes its own updates to Q
- Push-pull: P and Q send updates to each other

Observation

For push-pull it takes $\mathscr{O}(log(N))$ rounds to disseminate updates to all N nodes (round = when every node has taken the initiative to start an exchange).

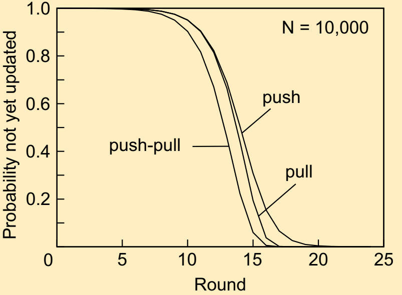
Anti-entropy: analysis

Basics

Consider a single source, propagating its update. Let p_i be the probability that a node has not received the update after the i^{th} round.

Analysis: staying ignorant

- With pull, $p_{i+1} = (p_i)^2$: the node was not updated during the i^{th} round and should contact another ignorant node during the next round.
- With push, $p_{i+1} = p_i(1 - \frac{1}{N-1})^{N(1-p_i)} \approx p_i e^{-1}$ (for small p_i and large N): the node was ignorant during the i^{th} round and no updated node chooses to contact it during the next round.
- With push-pull: $(p_i)^2 \cdot (p_i e^{-1})$



Gossip-based topology management

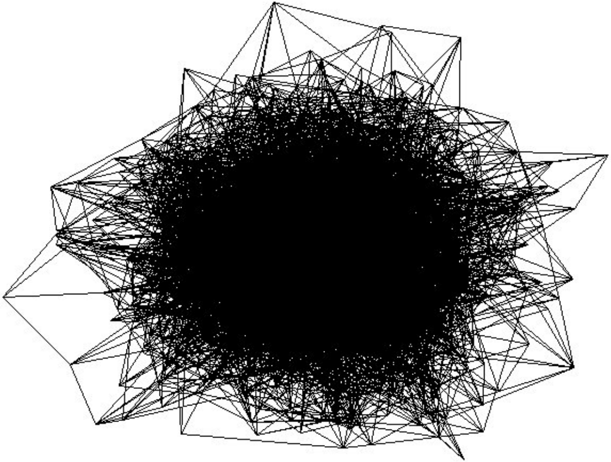
Largely based on joint work with

Mark Jelasity
(Univ. Szeged)

Spyros Voulgaris
(Univ. Athens)

Further reading by
M. Jelasity, A. Montresor, O. Babaoglu: “Gossip-based aggregation in large dynamic networks.” ACM TOCS, vol. 23(3) 2005.
M. Jelasity, S. Voulgaris, R. Guerraoui, A.M. Kermarrec, M. van Steen: “Gossip-based peer sampling” ACM TOCS, vol. 25(3) 2007.
S. Voulgaris, M. van Steen: “VICINITY: A Pinch of Randomness Brings out the Structure” Proc. Middleware, Lecture Notes in Computer Science, vol. 8275, pp. 21–40, Springer, 2013.

Constructing a grid



Basic approach

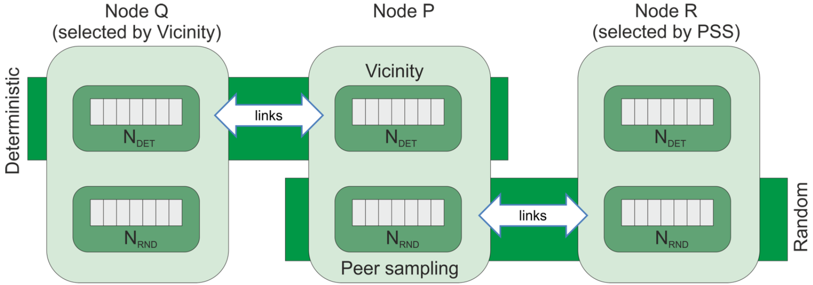
Essence

- Consider a set of networked nodes. Our goal is to construct an overlay network in a fully decentralized fashion.
- Every node maintains a partial view of the network: a few links to neighboring nodes.
- Each node periodically updates its view, by exchanging links with randomly selected neighbors.
- View updates changes a node's set of neighbors, but gradually drags the entire network to the required overlay structure.

Observation

It's already very difficult to understand this emerging behavior.

Vicinity with a peer-sampling service



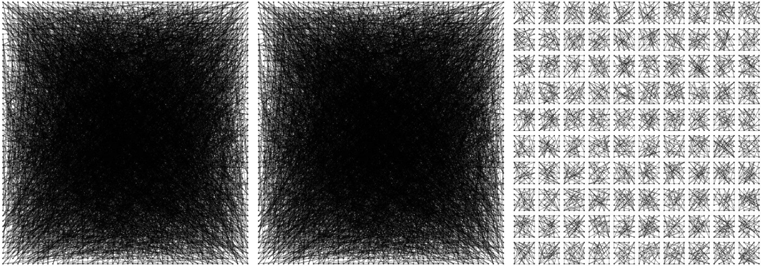
Two-layered partial view

- Random layer: randomly select a node from list, and then exchange *k* randomly selected links.
- Deterministic layer: exchange best links for realizing objective (such as a specific topology).

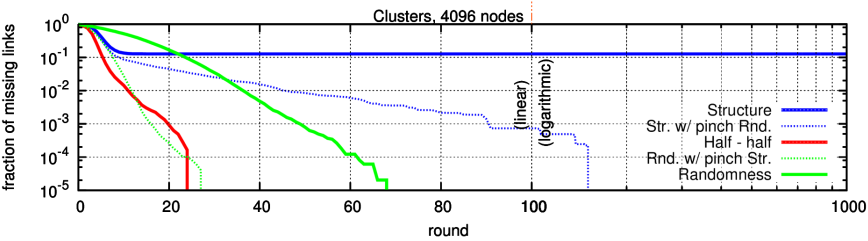
Example: Clustering nodes

Basics: Every node i is assigned a group identifier $GID(i) \in \mathbb{N}$. Our goal is to partition the overlay into disjoint components (clusters) such that

$$dist(i,j) = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are in the same group } [GID(i) = GID(j)] \\ 0 & \text{otherwise} \end{cases}$$



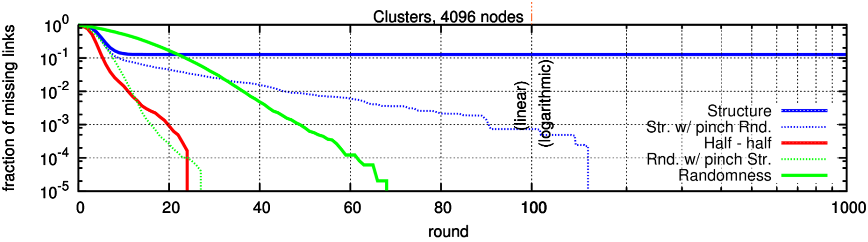
Example: Clustering nodes



Determinism versus Randomness

- Deterministic selection alone is a disaster
- Randomness is the determinant factor

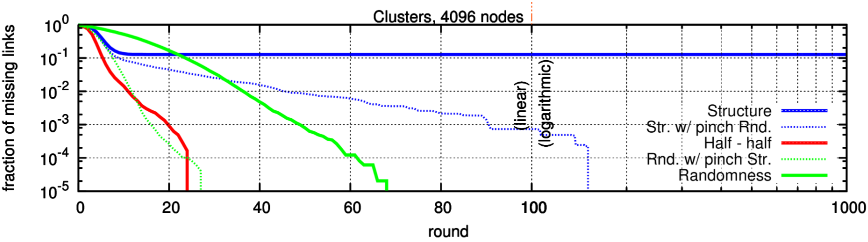
Example: Clustering nodes



Determinism versus Randomness

- Deterministic selection alone is a disaster
- ...but a pinch of randomness helps a lot
- Randomness is the determinant factor

Example: Clustering nodes



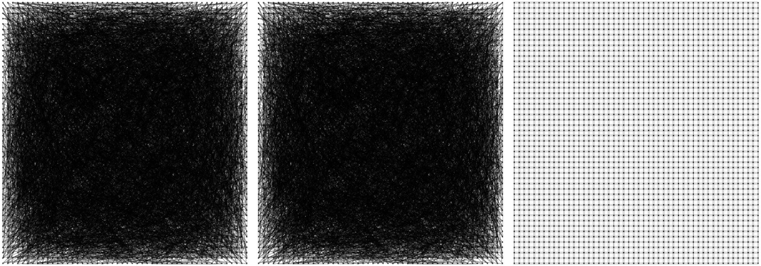
Determinism versus Randomness

- Deterministic selection alone is a disaster
- ...but a pinch of randomness helps a lot
- Randomness is the determinant factor
- The combination gives the best results

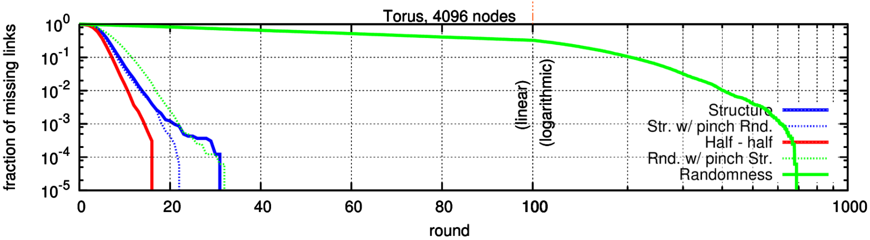
Example: 2D torus

Basics: Every node i has a position (x_i, y_i). Our goal is to link nodes to their closest neighbors in the Euclidean space.

$$\begin{aligned} dx = &\quad \min\{|x_i - x_j|, width - |x_i - x_j|\} \\ dy = &\quad \min\{|y_i - y_j|, height - |y_i - y_j|\} \\ dist(i,j) = &\quad \sqrt{dx^2 + dy^2} \end{aligned}$$



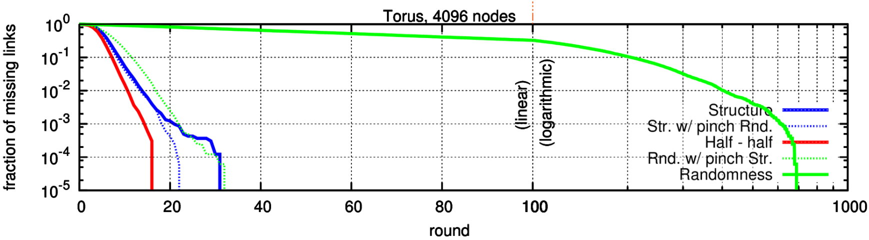
Example: 2D torus



Determinism versus Randomness

- Deterministic selection is the determinant factor
- Randomness alone is pretty bad

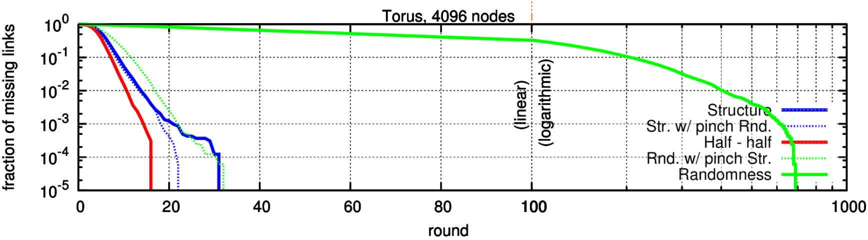
Example: 2D torus



Determinism versus Randomness

- Deterministic selection is the determinant factor
- ...but a pinch of randomness still helps
- Randomness alone is pretty bad

Example: 2D torus

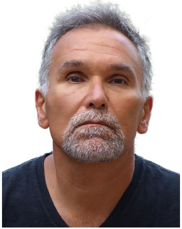


Determinism versus Randomness

- Deterministic selection is the determinant factor
- ...but a pinch of randomness still helps
- Randomness alone is pretty bad
- The combination gives the best results

Cognification in the feedback control loop

Largely based on work by

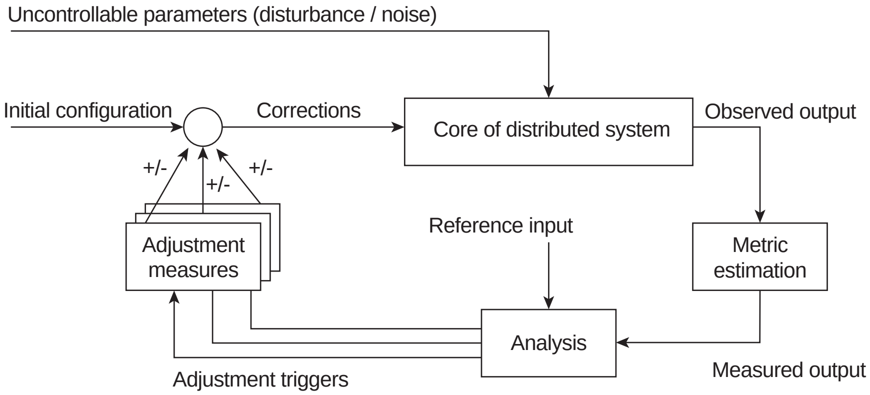


Ozalp Babaoglu
(Univ. Bologna)

Alina Sirbu
(Univ. Pisa)

Further reading:
A. Sibru, O. Babaoglu, A. Sirbu: “Towards operator-less data centers through data-driven, predictive, proactive autonomics,” Cluster computing, vol. 19, pp. 865-878, 2016.
O. Babaoglu, A. Sirbu: “Cognified distributed computing,” Proc. 37th Int’l Conf. on Distributed Computing Systems (ICDCS), 2018.

Distributed systems as feedback control loops



Predicting failing nodes in a data center

A simple fact

For many data centers, a human operator has on average tens of thousands of machines to monitor and to take action when things go wrong. Such numbers preclude massive scaling.

Predicting failing nodes in a data center

A simple fact

For many data centers, a human operator has on average tens of thousands of machines to monitor and to take action when things go wrong. Such numbers preclude massive scaling.

Observation

To properly maintain data centers at (future) warehouse scales, we need to make more room for autonomic computing in which operators can easily watch over 10- to 100-fold more machinery than is currently possible.

Predicting failing nodes in a data center

A simple fact

For many data centers, a human operator has on average tens of thousands of machines to monitor and to take action when things go wrong. Such numbers preclude massive scaling.

Observation

To properly maintain data centers at (future) warehouse scales, we need to make more room for autonomic computing in which operators can easily watch over 10- to 100-fold more machinery than is currently possible.

Starting point

Make sure that you can predict where things will go wrong before they do, so that measures can be automatically taken to sustain reliability (e.g., migrating virtual machines).

Data-driven predictive maintenance

Case study: source

A workload trace describing the status of nodes, jobs, and tasks of over 12,000 Google machines during a period of approximately 29 days. Events are recorded at 5-minute intervals, in total adding up to some 200 GB of data.

What to measure?

Task-based features

- Number of tasks running during the entire last 5 minutes
- Number of tasks started in the last 5 minutes
- Number of tasks that have finished in the last 5 minutes:
  - with status finished normally
  - with status evicted
  - with status failed
  - with status killed
  - with status lost (i.e., presumably terminated, but no record of the fact)

Node-based features

- Mean CPU usage rate
- Canonical memory usage (i.e., number of user-accessible pages)
- Mean disk I/O time
- Cycles per instruction (CPI)
- Memory accesses per instruction (MAI)

What to measure?

Expanding the feature set: aggregation

- For each basic feature, consider the six last time windows
- To observe deviation from the means, measure mean, deviation, and coefficient of variation (σ/μ) over resolutions: 1, 12, 24, 48, 72, 96 hours.

Cross-correlation features

- Look at the correlation between # running, started, failed jobs; CPU load; memory usage; disk I/O; CPI.
- Consider all possible $\binom{7}{2} = 21$ combinations for each of the 6 resolutions.

Measuring cluster behavior

- Per node: uptime
- Number of nodes that failed in the past hour

Total feature set

Total number of features to look at

- 7 plus 5 basic features, measured for 6 time windows
- Uptime per node
- Number of failed nodes in past hour
- 6 different resolutions for each basic feature
- 3 statistics per resolution
- 21 correlations

End result

Adds up to ($6 \times 12 = 72$ plus 2 features) plus ($3 \times 6 \times 12 = 216$ plus $21 \times 6 = 126$) aggregation features, in total 416 features.

Total size of the data set: over 100 million data points.

The data science

Essence

We're dealing with a classification problem that requires predicting whether a node will fail, based on what we know so far. What is needed is a training data set that can be used to build a classifier.

Data set

The data set consists of relatively few failure detections in comparison to nonfailing nodes, creating an imbalance. Therefore, consider a random sample of 0.5% data points representing nonfailing nodes and bringing the two types in balance in the final data set.

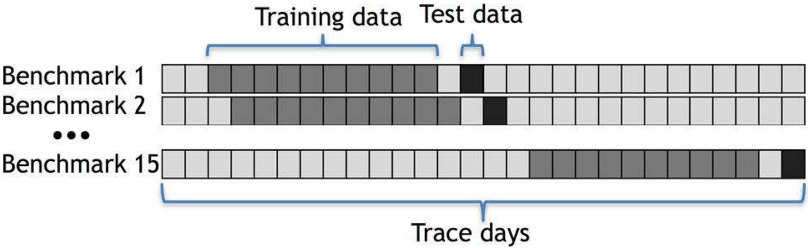
⇒ a data set with over 650,000 data points.

The data science

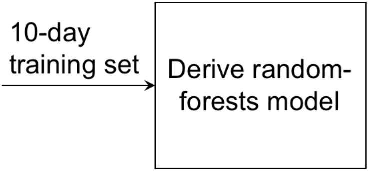
Method

Many data points correspond to the same event (i.e., represent the same failure), so random sampling to construct a training set and a subsequent test set may show too many overlaps leading to nonrealistic optimal results.

⇒ split data according to time.

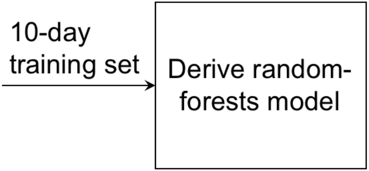


The data science

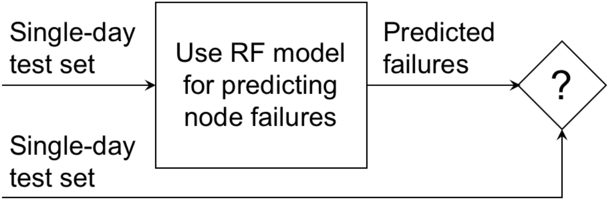


Model building

The data science



Model building



Model testing

Displaying results

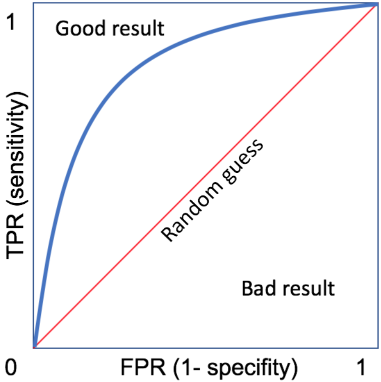
Understanding receiver operating characteristic (ROC)

- Any binary classifier will lead to true positives and false positives when used for predictions.
- Assume that the classifier is dependent on a parameter s^*.
- The true positve rate (TPR) is now a function of s^*
- The false positve rate (FPR) is now a function of s^*
- By varying s^*, we can obtain a scatterplot of (TPR,FPR) values

Displaying results

Understanding receiver operating characteristic (ROC)

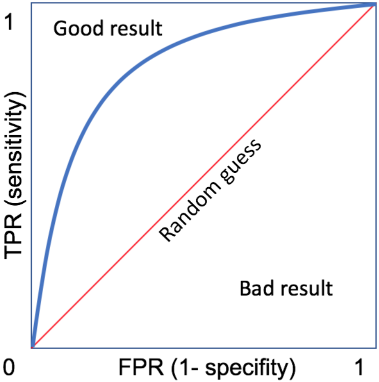
- Any binary classifier will lead to true positives and false positives when used for predictions.
- Assume that the classifier is dependent on a parameter s^*.
- The true positve rate (TPR) is now a function of s^*
- The false positve rate (FPR) is now a function of s^*
- By varying s^*, we can obtain a scatterplot of (TPR,FPR) values



Displaying results

Understanding receiver operating characteristic (ROC)

- Any binary classifier will lead to true positives and false positives when used for predictions.
- Assume that the classifier is dependent on a parameter s^*.
- The true positve rate (TPR) is now a function of s^*
- The false positve rate (FPR) is now a function of s^*
- By varying s^*, we can obtain a scatterplot of (TPR,FPR) values



Understanding precision

The true positive rate (TPR) is also known as the recall. The precision is the fraction of true positives over all output (= true and false positives).

The results

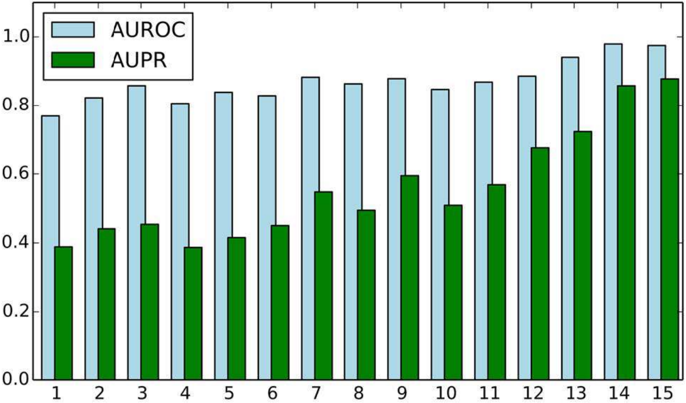
The dependent parameter s^*

For each node j, a score s_j was computed that was proportional to the liklihood that j would belong to the class of failing nodes. Then, a threshold s^* was used, such that if $s_j \geq s^*$, node j would be predicted to fail.

The results

The dependent parameter s^*

For each node j, a score s_j was computed that was proportional to the liklihood that j would belong to the class of failing nodes. Then, a threshold s^* was used, such that if $s_j \geq s^*$, node j would be predicted to fail.



Cognification: conclusions

Important results

- Using appropriate datasets (i.e., including proper sampling), high-quality predictions on failing nodes can be obtained.
- Based on the given dataset and experiment, data-driven predictive maintenance is a highly promising approach.
- A similar approach has been shown to work for predicting energy consumption.

Cognification: conclusions

Important results

- Using appropriate datasets (i.e., including proper sampling), high-quality predictions on failing nodes can be obtained.
- Based on the given dataset and experiment, data-driven predictive maintenance is a highly promising approach.
- A similar approach has been shown to work for predicting energy consumption.

Conclusion

Data-driven analytics without seeing all the gory details will allow us to partly move from classical fault tolerance to fault prevention by simply replacing components before they break.

Take-away messages

Lessons

- There are matters that are simply complicated, such as reaching consensus in distributed systems.
- There are many matters that can be kept simple, such as multicasting by flooding & gossiping.
- Distributed systems are entering a new era by taking data analysis into account. Cognification will allow us to draw conclusions without knowing all the gory details.

Take-away messages

Lessons

- There are matters that are simply complicated, such as reaching consensus in distributed systems.
- There are many matters that can be kept simple, such as multicasting by flooding & gossiping.
- Distributed systems are entering a new era by taking data analysis into account. Cognification will allow us to draw conclusions without knowing all the gory details.

Crucial observations that are too often ignored

- It is relatively simple to invent complicated solutions, yet it may be very difficult to invent simple ones.

Take-away messages

Lessons

- There are matters that are simply complicated, such as reaching consensus in distributed systems.
- There are many matters that can be kept simple, such as multicasting by flooding & gossiping.
- Distributed systems are entering a new era by taking data analysis into account. Cognification will allow us to draw conclusions without knowing all the gory details.

Crucial observations that are too often ignored

- It is relatively simple to invent complicated solutions, yet it may be very difficult to invent simple ones.
- Discard thoughts on optimizations until you fully understand the problem at hand.