

Distributed Systems

Advanced Course

2. Basic Abstractions



Paris Carbone

A SYSTEM'S ROADMAP

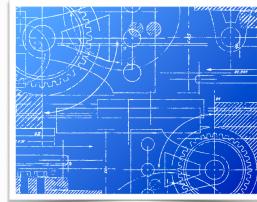
I- Specification



The ‘WHAT’

- *Assumptions*
- *Goals*
- *Set of Properties*

II- Solution Design



The ‘HOW’

- *Satisfies Properties*
- *Abstract yet Accurate Representation*

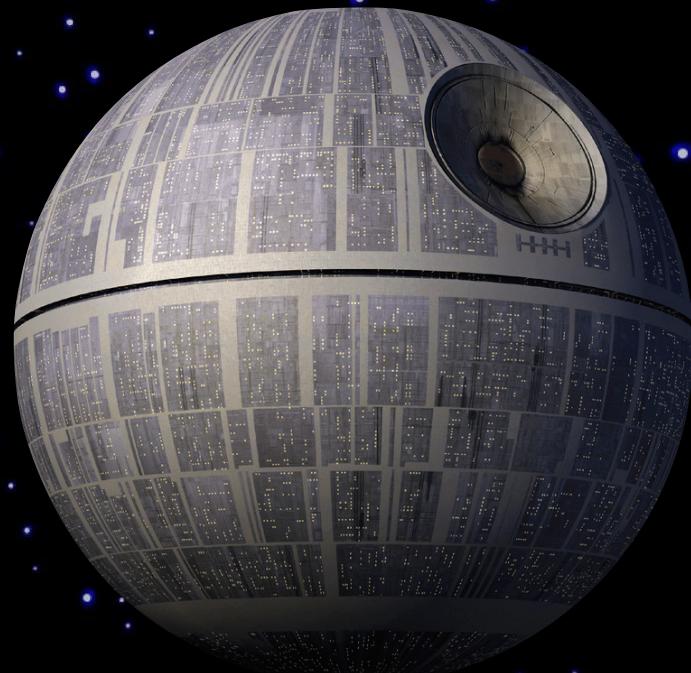
III- Implementation



- *Execution*
- *Development*

**Let's take a closer look into
...one of the biggest systems of all time**

The Death Star



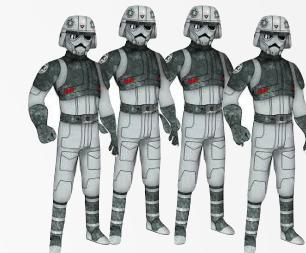
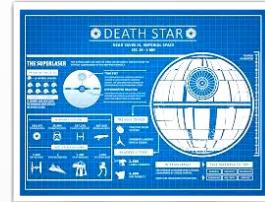
DEATH STAR ROADMAP



I- Specification



II- Solution Design

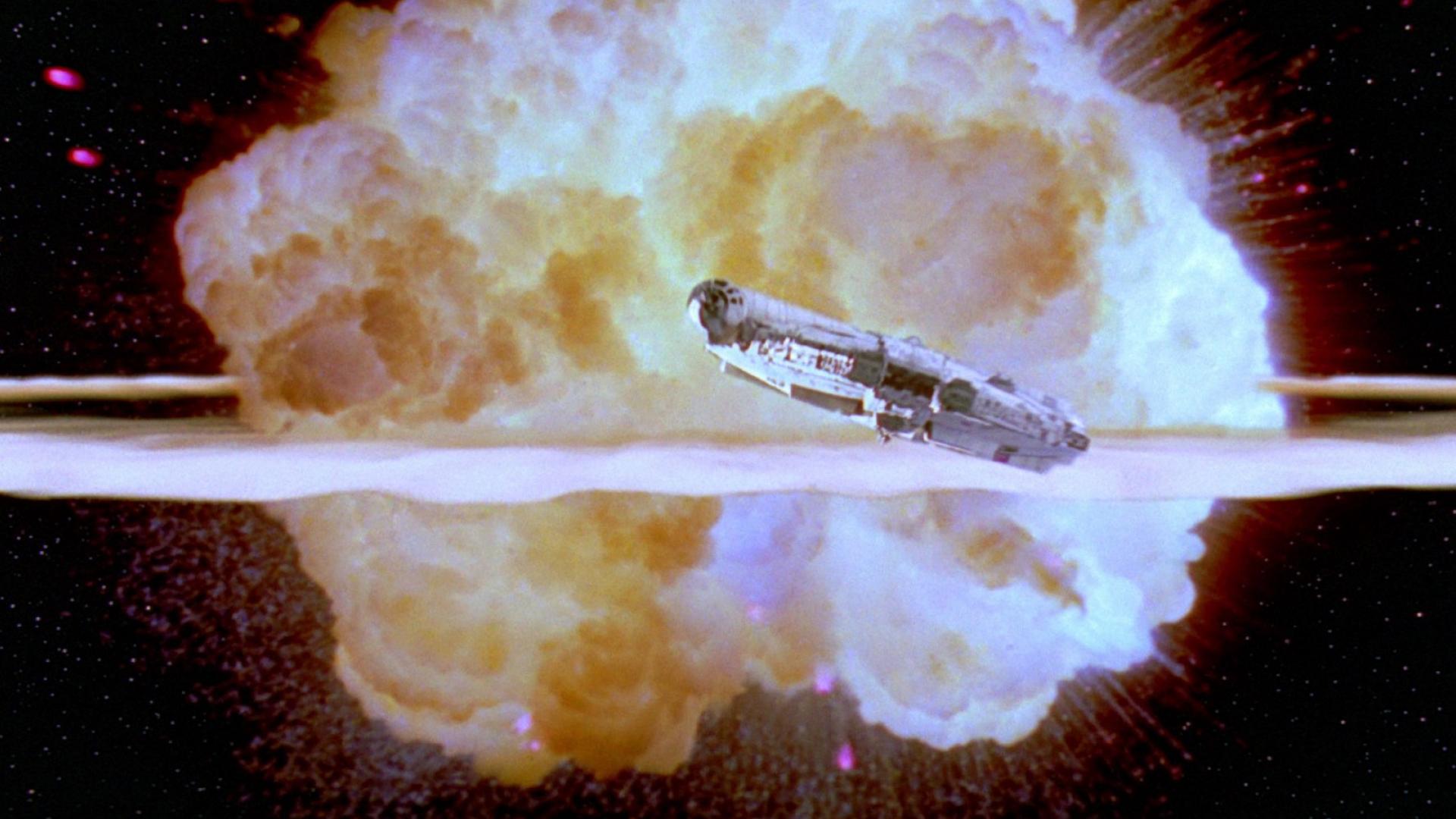


III- Implementation



- Gargantuan Scale/Storage
- Indestructible
- Ultra High-Speed (>light)
- Massive Power Projection

- Moon-Size Model
- Stainless Steel Plates
- Hyperdrive, Thermal Reactors
- Superlaser Module Design



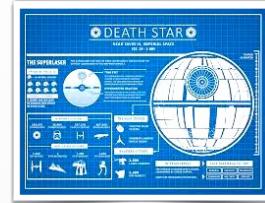
THE ISSUE



I- Specification



II- Model (Blueprint)



shoot here to
detonate



- Gargantuan Scale/Storage
- **Indestructible**
- Ultra High-Speed (>light)
- Massive Power Projection
- Moon-Size Model
- Stainless Steel Plates
- Hyperdrive, **Thermal Reactors**
- Superlaser Module Design

WE COULD HAVE SAVED DEATH STAR

- As with every type of reliable system
 1. A correct, careful specification of its properties is crucial.
 2. A solution design (algorithm) needs to:
 1. Provably satisfy all properties and
 2. Not violate any property (duh).

Let's see how this can be done with some core abstractions!

COURSE TOPICS



- ▶ Intro to Distributed Systems
- ▶ Fundamental Abstractions and Failure Detectors
- ▶ Reliable and Causal Order Broadcast
- ▶ Consensus (Paxos)
- ▶ Replicated State Machines (OmniPaxos, Raft, Zab etc.)
- ▶ Distributed Shared Memory-CRDTs
- ▶ Time Abstractions and Interval Clocks (Spanner etc.)
- ▶ Consistent Snapshotting (Stream Data Management)
- ▶ Distributed ACID Transactions (Cloud DBs)

NEED OF DISTRIBUTED ABSTRACTIONS

*Reliable applications need underlying services
stronger than network protocols (e.g. TCP, UDP)*

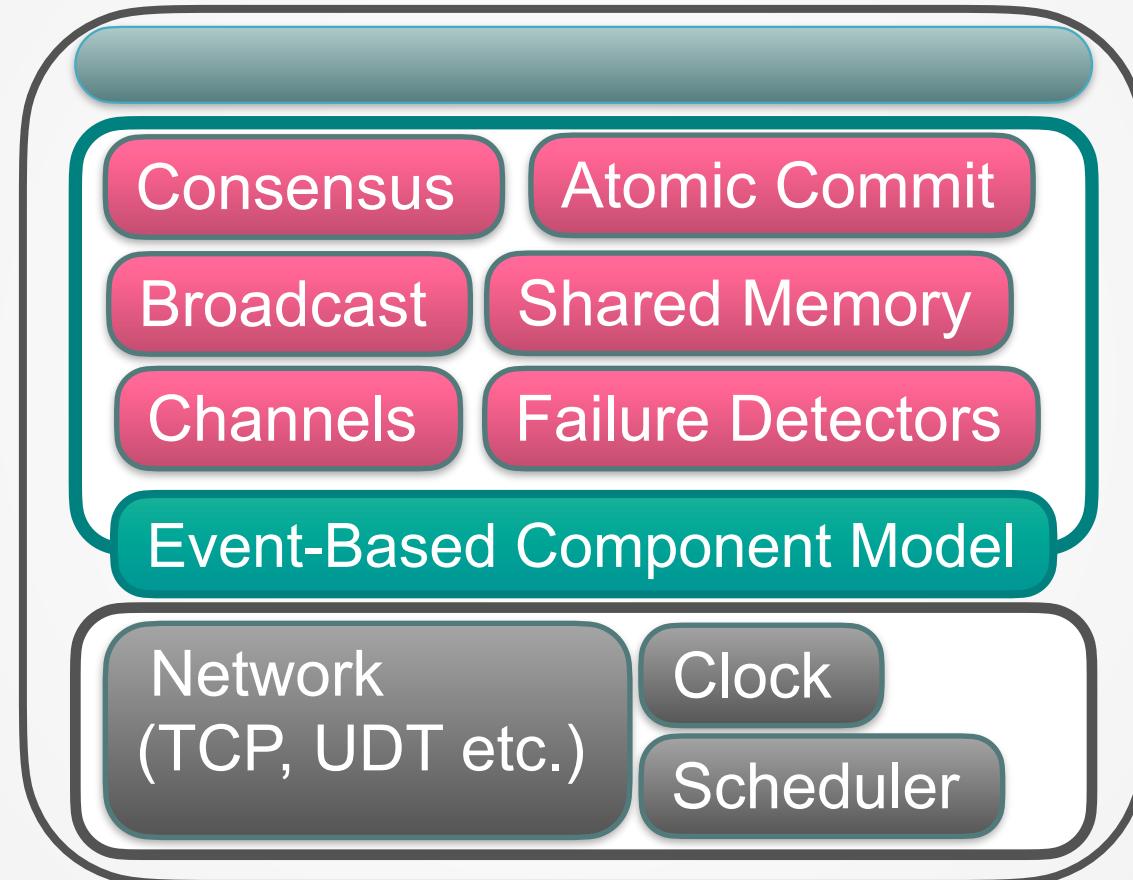
- The basic building blocks of **any** distributed system is a **set of distributed algorithms**.
- Implemented as a **middleware** between network (OS) and the application.

ANATOMY OF A DISTRIBUTED SYSTEM

Distributed Applications

Middleware

OS

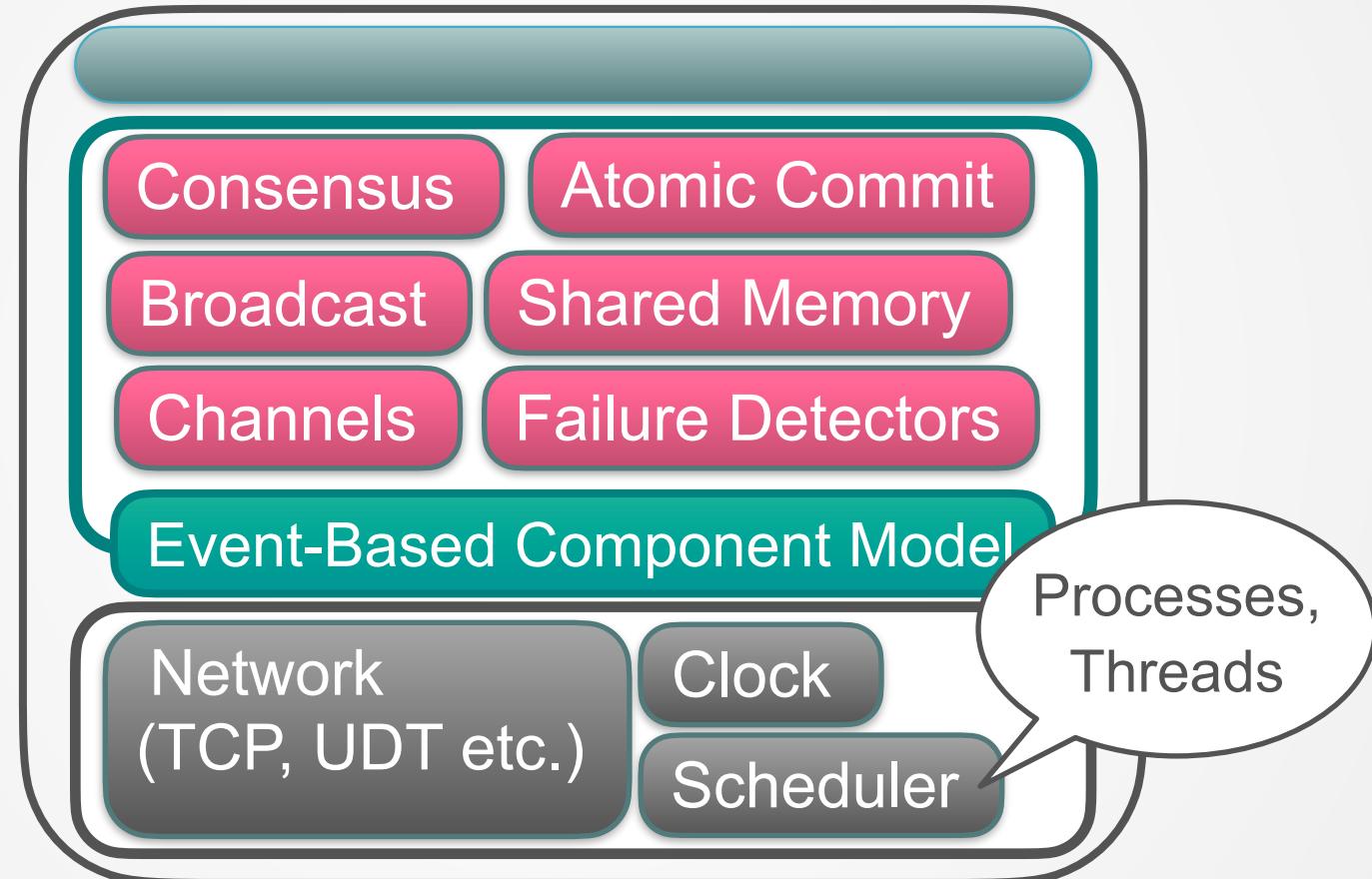


ANATOMY OF A DISTRIBUTED SYSTEM

Distributed Applications

Middleware

OS

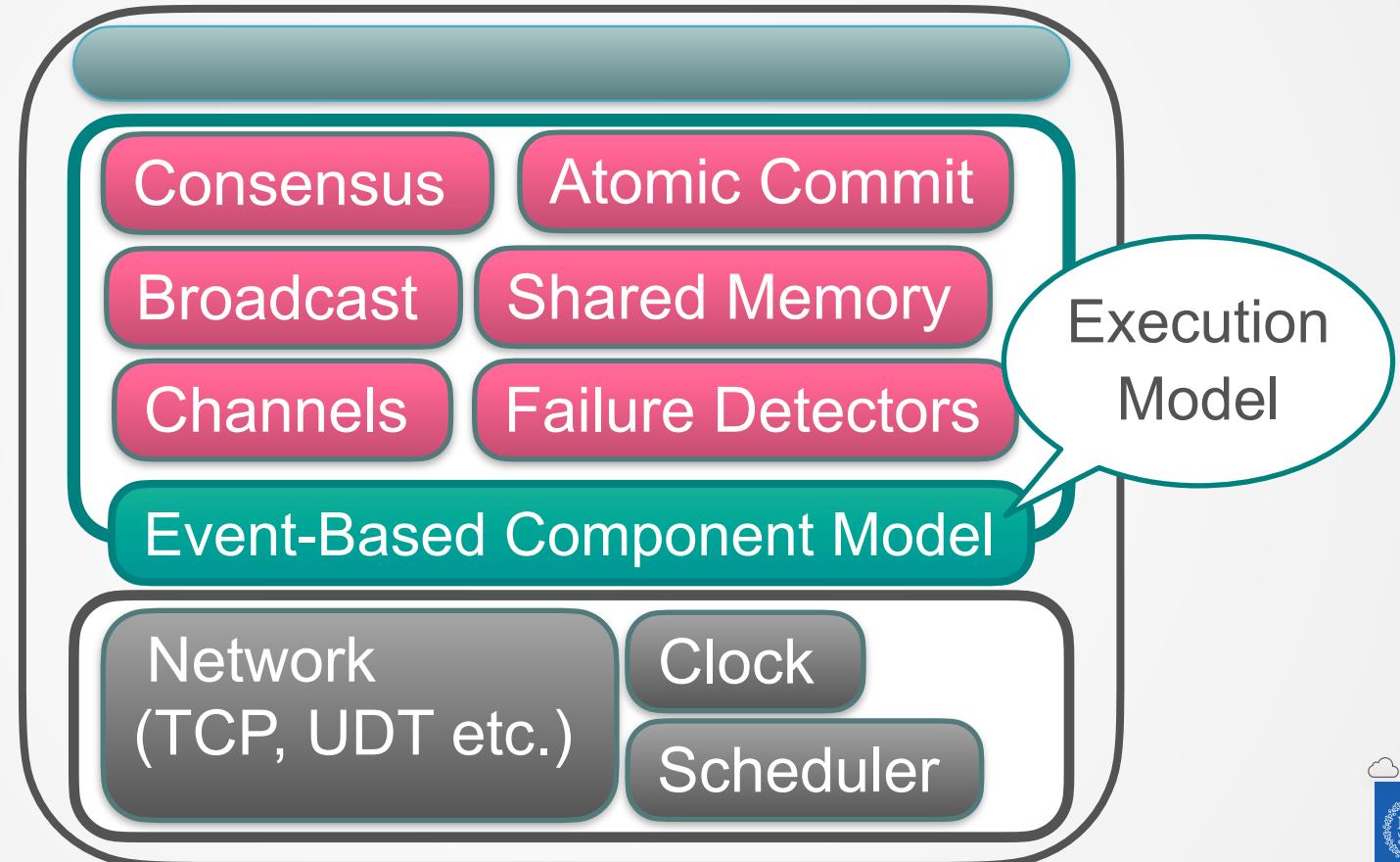


ANATOMY OF A DISTRIBUTED SYSTEM

Distributed Applications

Middleware

OS

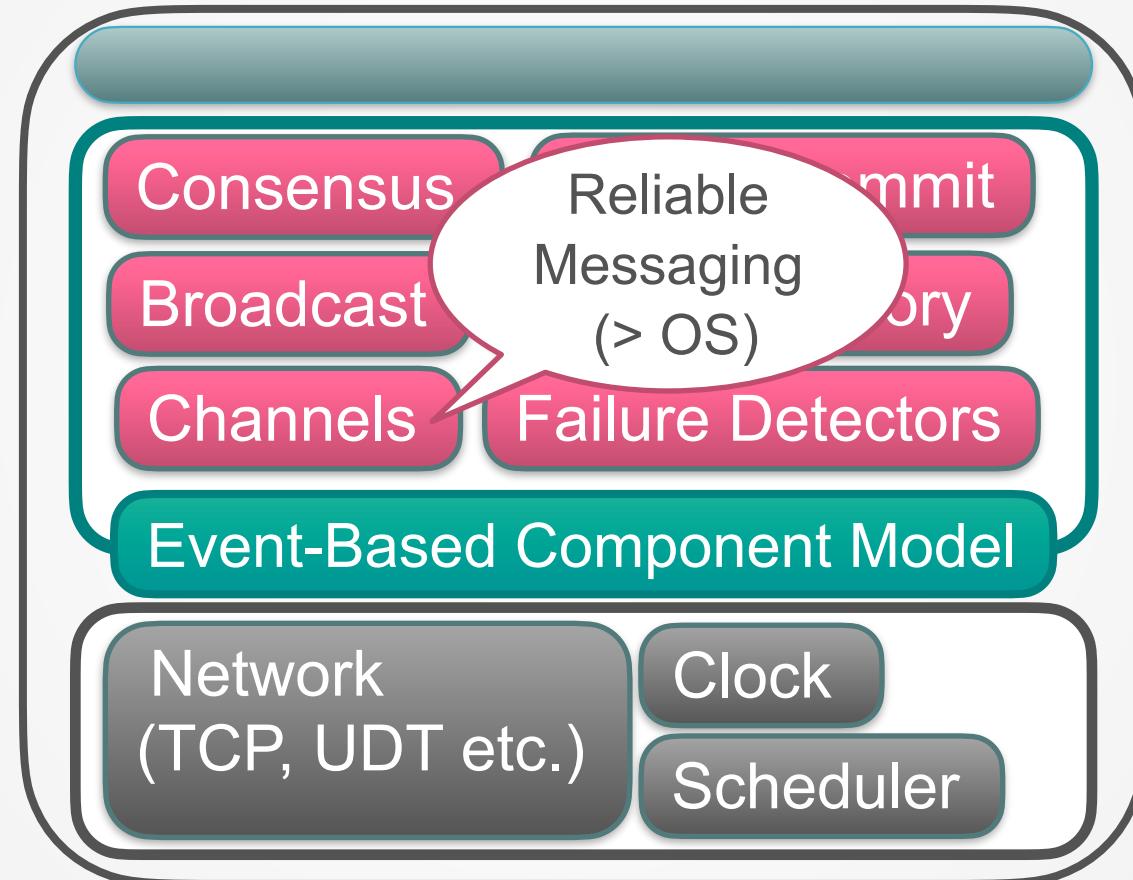


ANATOMY OF A DISTRIBUTED SYSTEM

Distributed Applications

Middleware

OS

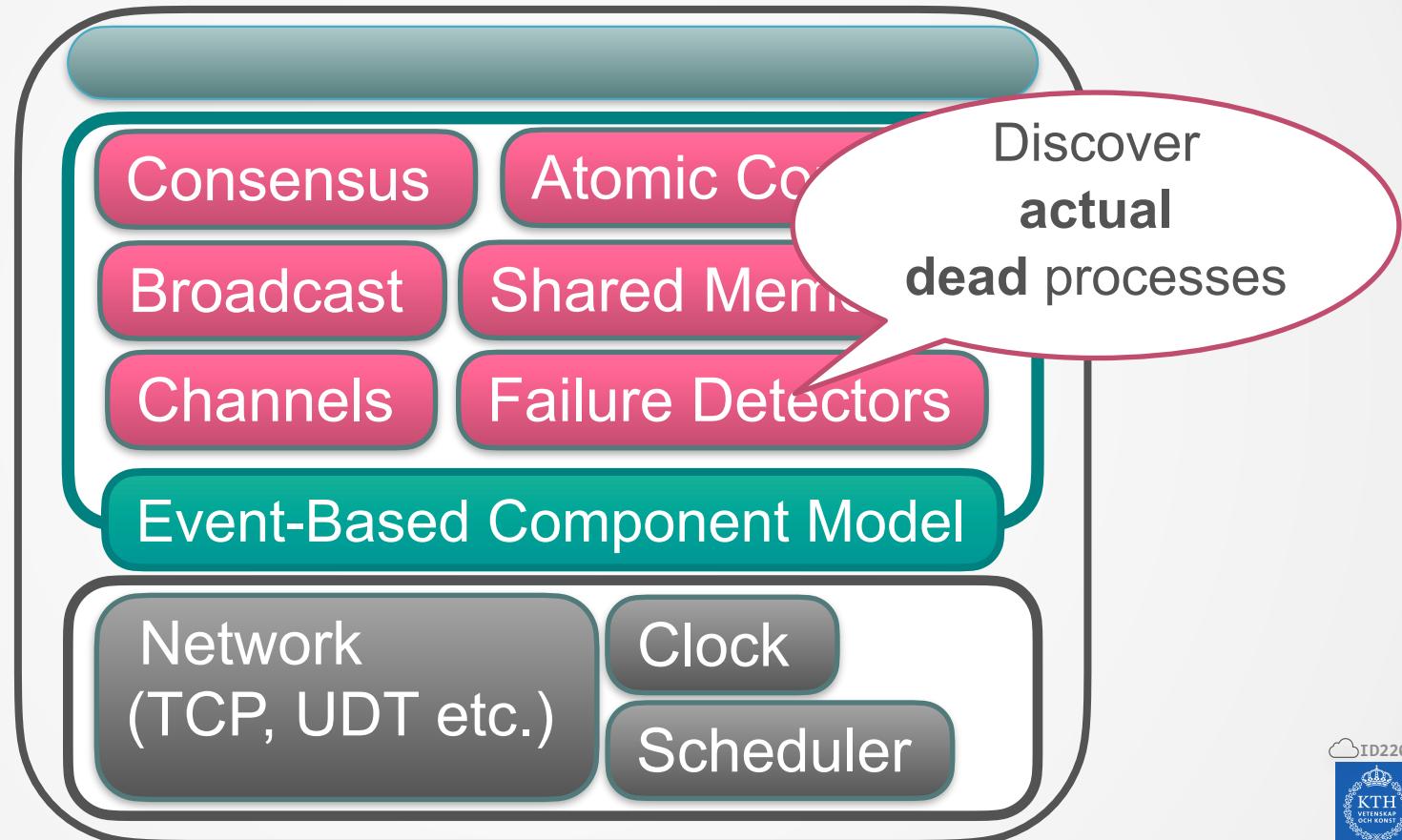


ANATOMY OF A DISTRIBUTED SYSTEM

Distributed Applications

Middleware

OS

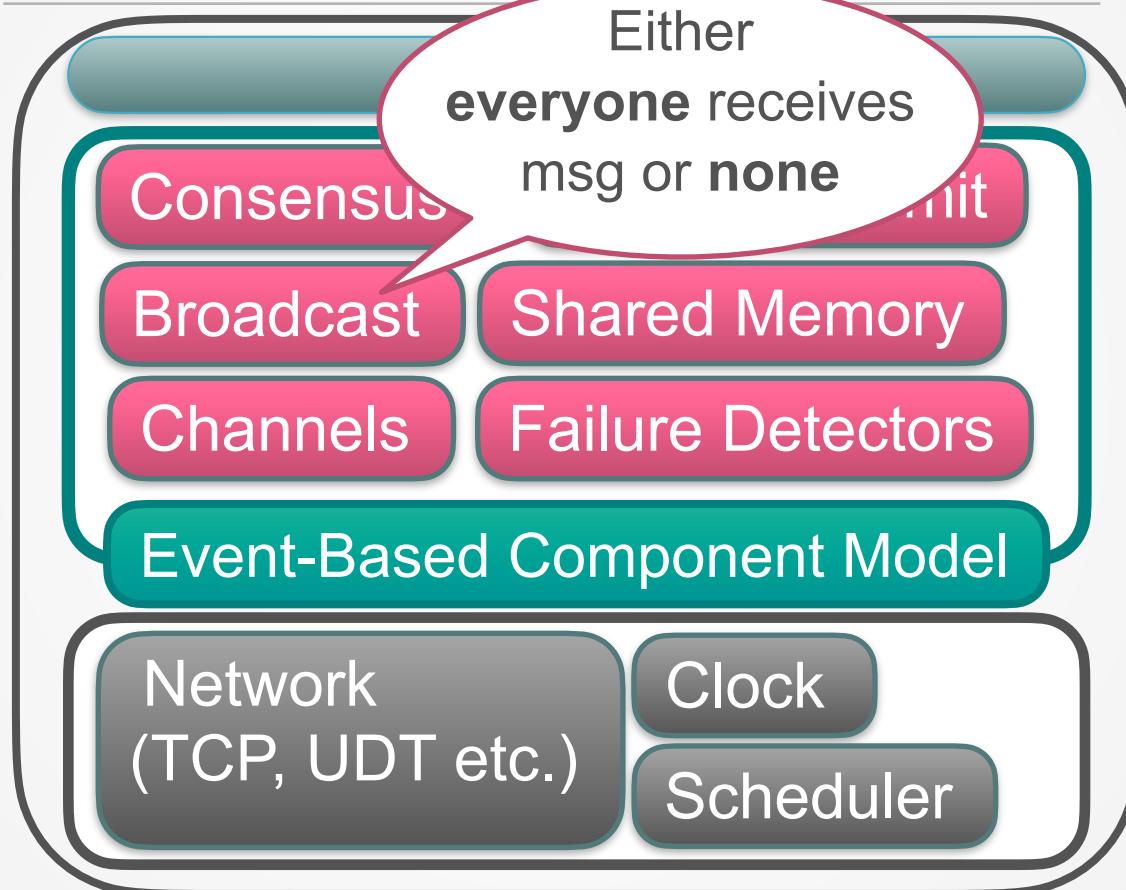


ANATOMY OF A DISTRIBUTED SYSTEM

Distributed Applications

Middleware

OS

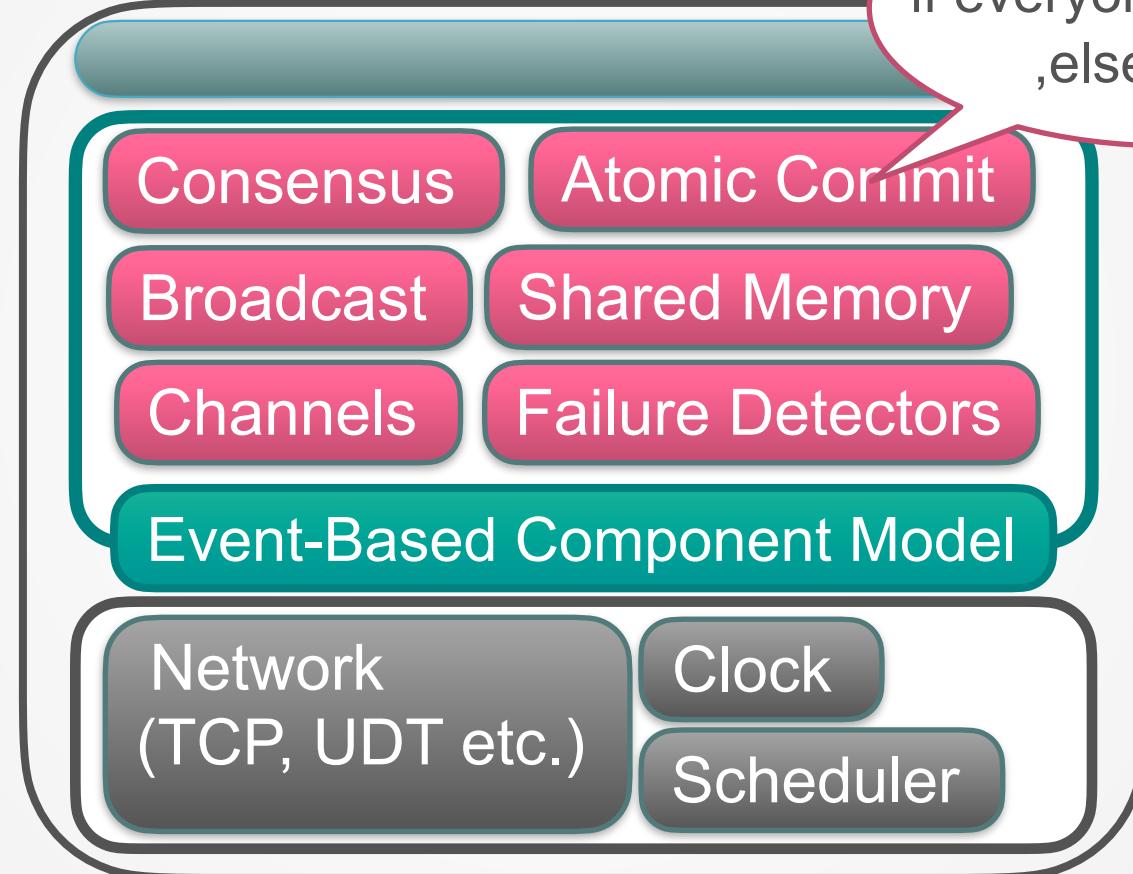


ANATOMY OF A DISTRIBUTED SYSTEM

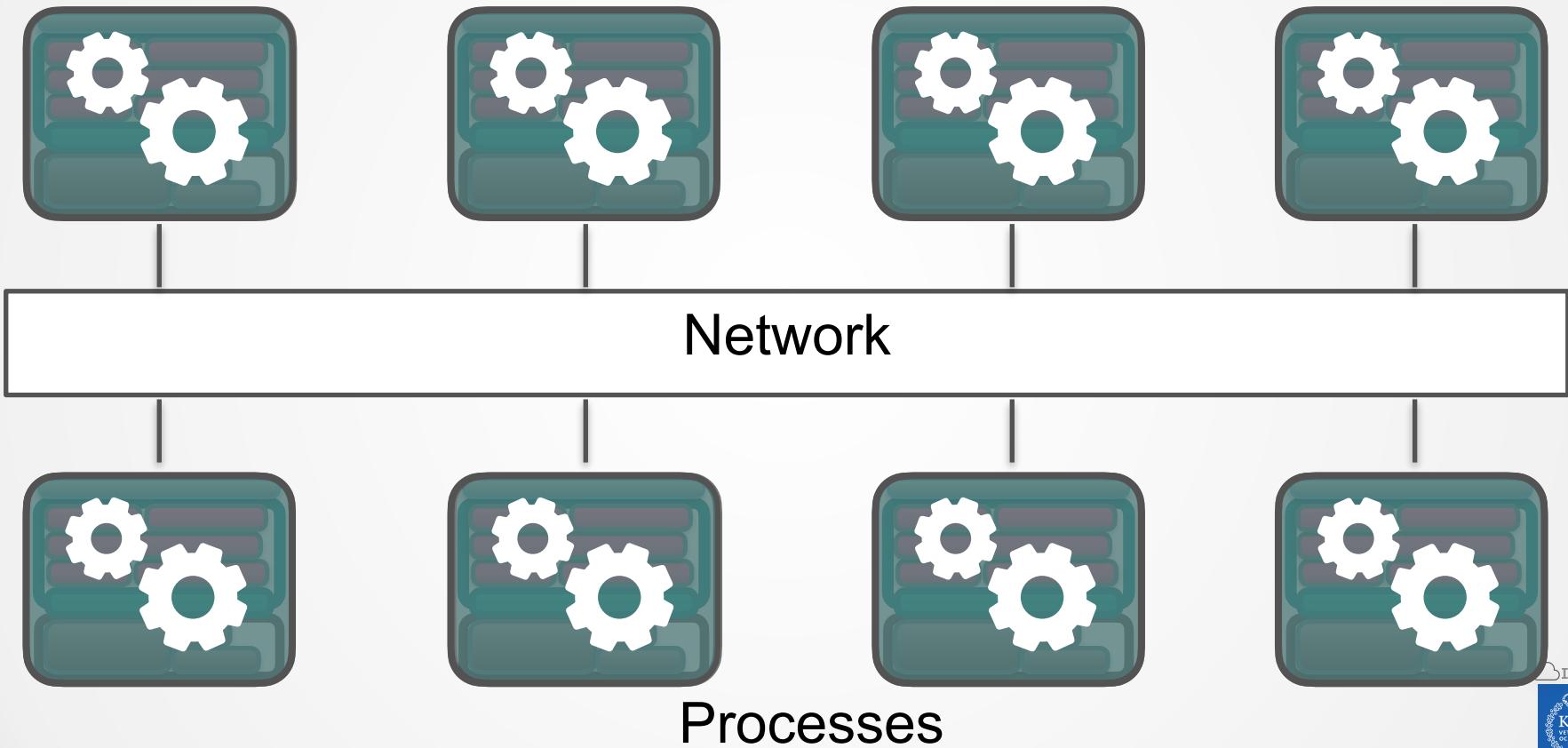
Distributed Applications

Middleware

OS



ANATOMY OF A DISTRIBUTED SYSTEM



The Event-based Component Model



Event-Based Component Model

Network
(TCP, UDT etc.)

Consensus

Atomic Commit

Channels

Shared Memory

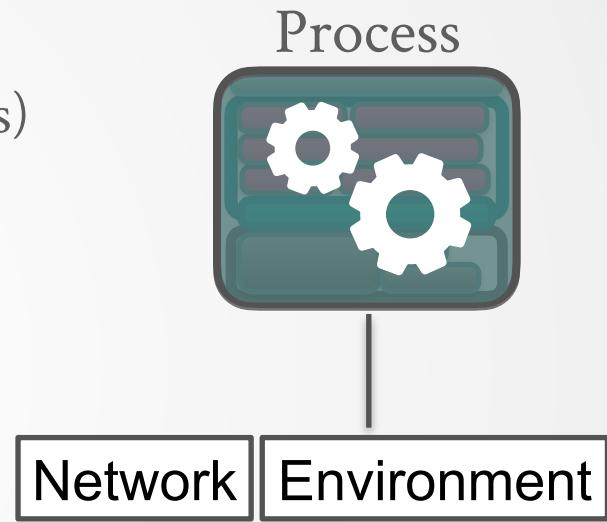
Failure Detectors

Clock

Scheduler

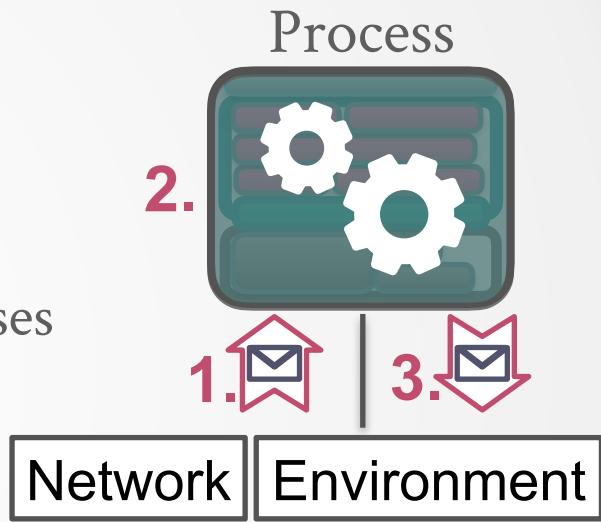
DISTRIBUTED COMPUTING MODEL

- Set of **processes** and a **network** (communication links)
- Each process runs a **local algorithm** (program)
- Each process makes **computation steps**
- The network makes computation **steps**
 - to store a message sent by a process
 - to deliver a message to a process
- Message delivery **triggers** a computation step at the receiving process



THE DISTRIBUTED COMPUTING MODEL

- Computation step at a process
 1. Receives a message
 2. Performs local computation and updates state
 3. Sends one or more messages to some other processes
- Communication step:
 - Depends on the network abstraction
 - Receives a message from a process, or
 - Delivers a message to a process





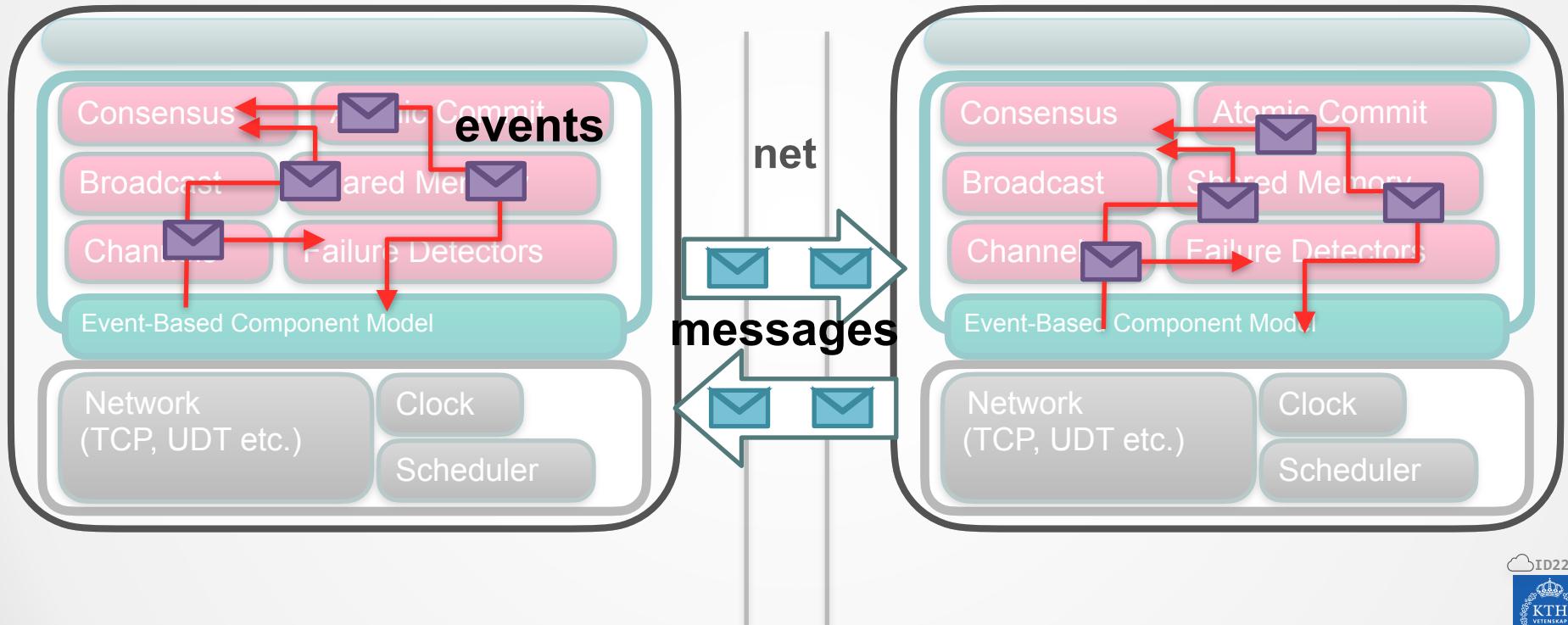
INSIDE A PROCESS

- A process consists of a set of components (automata)
- Components are **concurrent** and access local **state**.
- Each component receives messages through an input **FIFO buffer**
- Sends messages to other components



- **Events**: messages between components in the same process
- Events are handled by procedures (actions) called [Event Handlers](#)

EVENTS VS MESSAGES



EVENT-BASED PROGRAMMING

- Process executes program
 - Each program consists of a set of **modules** or **component specifications**
 - At runtime these are deployed as **components**
 - The components in general form a software stack

EVENT-BASED PROGRAMMING

Process executes program

Components interact via **events** (with attributes):

Handled **asynchronously** by Event Handlers

```
on event <co, Event1, attr1, attr2,...> do
    // local computation
    trigger <co, Event2, attr3, attr4,...>
```

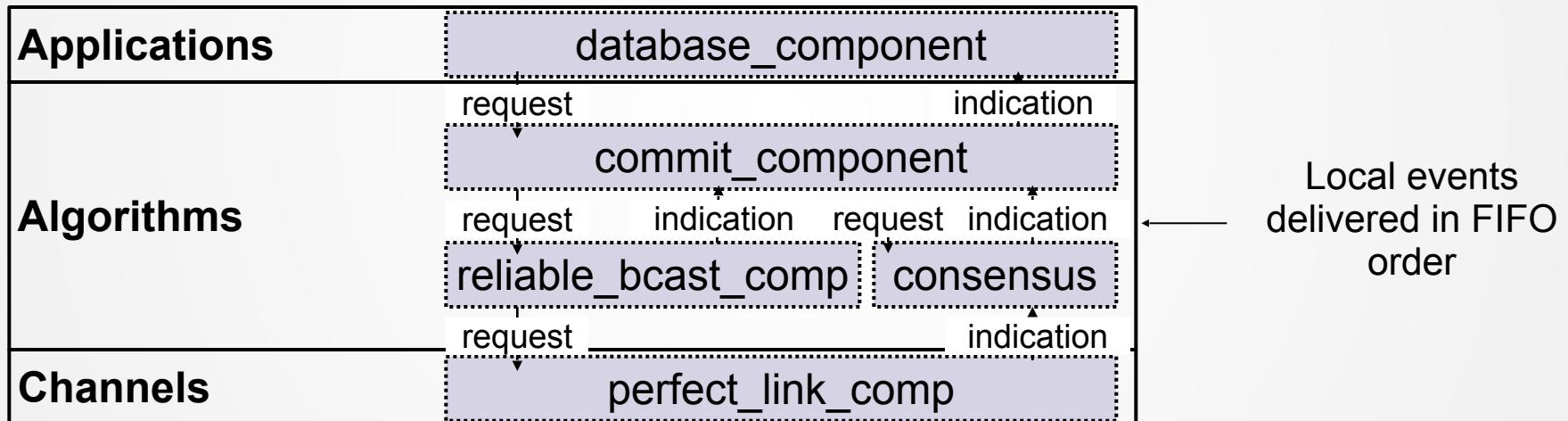
EVENT-BASED PROGRAMMING

- Events include any scheduled action
 - New Message (most of the time)
 - A Timeout (from Environment Clock)
 - A State Condition (e.g. $x==5$ & $y<9$)
- Two types of events
 - Requests (incoming to component)
 - Indications (outgoing from component)

COMPONENTS IN A PROCESS



Example stack of **components** in a single process



CHANNELS AS COMPONENTS

Channels represented by components (too)

Request (input) event:

Send to destination some message (with data)

trigger <send | dest, [data1, data2, ...] >

< header/type | parameters/payload >

Indication (output) event:

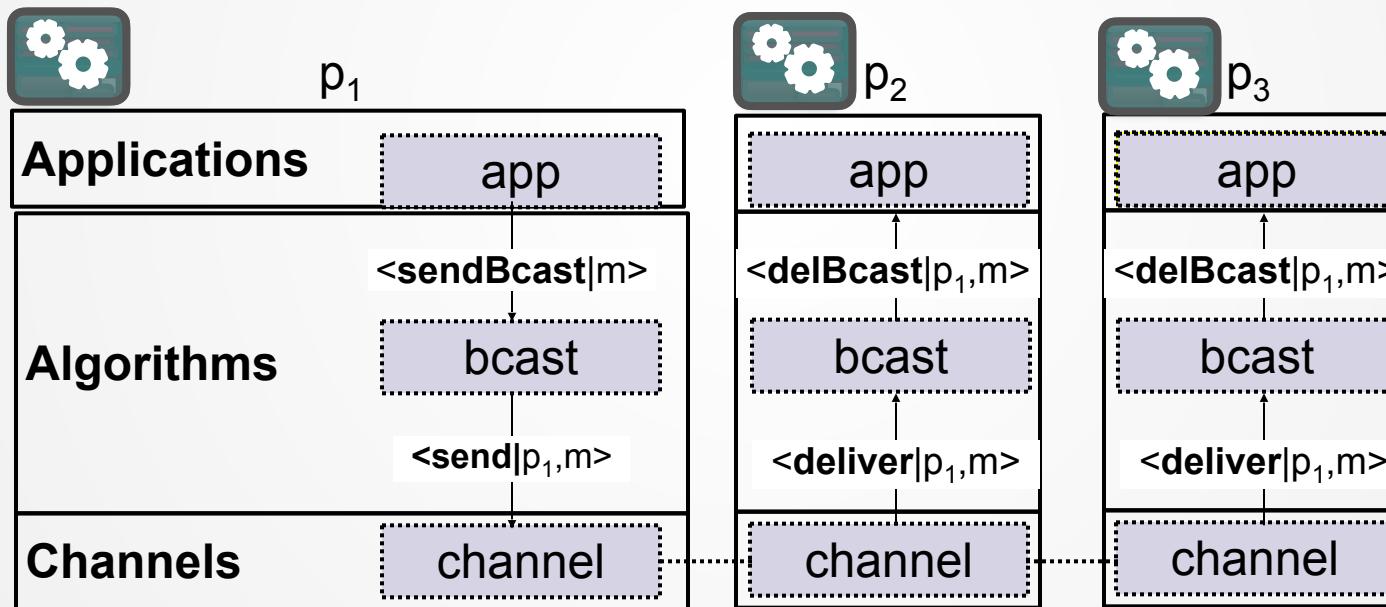
Deliver from source some message (with data)

upon event <deliver | src, [data1,data2, ...]> do

EXAMPLE

Application uses a Broadcast component

which uses channel component to broadcast



Specifications



SPECIFICATION OF A SERVICE



How to specify a distributed service (abstract)?

1. Interface (aka Contract, API)

Requests
Indications/Responses

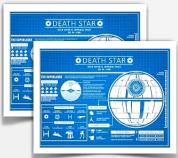
2. Correctness Properties

Safety
Liveness

3. Underlying Model

Assumptions on failures
Assumptions on timing (amount of synchrony)

declarative specification
“what”
aka problem



Implementation

Composed of other services
Adheres to interface and satisfies correctness
Has internal events

imperative,
many possible
“how”





SIMPLE EXAMPLE: JOB HANDLER

how to use

conditions

Module:

Name: JobHandler, instance *jh*

Events:

Request: $\langle jh, \text{Submit} \mid job \rangle$: Requests a job to be processed

Indication: $\langle jh, \text{Confirm} \mid job \rangle$: Confirms that the given job has been (or will be) processed

Properties:

Guaranteed response: Every submitted job is eventually confirmed



SOLUTION EXAMPLE

Synchronous Job Handler

Implements:

JobHandler, **instance *jh***

upon event $\langle jh, \text{Submit} \mid job \rangle$ **do**

process(*job*)

trigger $\langle jh, \text{Confirm} \mid job \rangle$



ANOTHER SOLUTION: ASYNCHRONOUS JOB HANDLER

Implements:

```
JobHandler, instance jh
upon event ⟨jh, Init⟩ do
    buffer := ∅
```

⟨..*Init*⟩ automatically
generated upon component
creation

```
upon event ⟨jh, Submit | job⟩ do
    buffer := buffer ∪ {job}
    trigger ⟨jh, Confirm | job⟩

upon buffer ≠ ∅ do
    job := selectjob (buffer)
    process(job)
    buffer := buffer \ {job}
```

Safety and Liveness Properties

SPECIFICATION OF A SERVICE

How to specify a distributed service (abstract)?

Interface (aka Contract, API)

Requests

Responses

Correctness Properties

Safety

Liveness

Model

Assumptions on failures

Assumptions on timing (amount of synchrony)

Implementation

Composed of other services

Adheres to interface and satisfies correctness

Has internal events

declarative specification

“what”

aka problem

imperative,
many possible
“how”

SPECIFYING CORRECTNESS

Safety

Properties that state that nothing bad ever happens

Liveness

Properties that state that something good eventually happens

CORRECTNESS EXAMPLE

- Correctness of **Yourself** in ID2203

Safety

You should **never** fail the exam

(marking exams costs money)

Liveness

You should **eventually** take the exam

(university gets money when you pass)

CORRECTNESS EXAMPLE (2)

- Correctness of traffic lights at intersection

Safety

Only one direction should have a green light



Liveness

Every direction should *eventually* get a green light

EXECUTION AND TRACES

An **execution fragment** of algorithm A is sequence of alternating states and events

$s_0, \varepsilon_1, s_1, \varepsilon_2, \dots, s_r, \varepsilon_r, \dots$

$(s_k, \varepsilon_{k+1}, s_{k+1})$ **transition** of A for $k \geq 0$

An **execution** is execution fragment where s_0 is an initial state

A **trace** of an execution E, $\text{trace}(E)$

The subsequence of E consisting of **all external** events

$\varepsilon_1, \varepsilon_2, \dots, \varepsilon_r, \dots$

SAFETY & LIVENESS ALL THAT MATTERS

A trace **property** P is a function that takes a trace and returns true/false
i.e. P is a **predicate**

Any trace property can be expressed as the conjunction of a **safety** property and a **liveness** property”

SAFETY FORMALLY DEFINED

The **prefix** of a trace T is the first k (for $k \geq 0$) events of T

I.e. cut off the tail of T

I.e. finite beginning of T

An **extension** of a prefix P is any trace that has P as a prefix

SAFETY DEFINED

Informally, property P is a **safety property** if

Every trace T violating P has **a bad event**, s.t.
every execution starting like T and behaving like
T up to the bad event (including), will violate P
regardless of what it does afterwards

SAFETY DEFINED

Formally, a property P is a **safety** property if

Given any execution E such that $P(\text{trace}(E)) = \text{false}$,

There exists a prefix of E, s.t. every extension of that prefix gives an execution F s.t. $P(\text{trace}(F))=\text{false}$

SAFETY EXAMPLE

Point-to-point message communication

Safety P: “At most once delivery”

A message sent is delivered **at most** once

SAFETY EXAMPLE

Point-to-point message communication

Safety P: “At most once delivery”

A message sent is delivered **at most** once

Take an execution where a message is delivered more than once

- Cut-off the tail after the second delivery
- Any continuation (extension) will give an execution which also violates the required property

LIVENESS FORMALLY DEFINED

- A property P is a **liveness** property if
 - Given any prefix F of an execution E,
there exists an extension of $\text{trace}(F)$ for
which P is true

“As long as there is life there is hope”

LIVENESS EXAMPLE

Point-to-point message communication

Liveness P: “At least once delivery”

A message sent is delivered **at least** once

Take the prefix of any execution

- If prefix contains delivery, any extension satisfies P
- If prefix doesn’t contain the delivery, extend it so that it contains a delivery, the prefix + extended part will satisfy P

MORE ON SAFETY

Safety can only be

satisfied in infinite time (you're never safe)

violated in finite time (when the bad happens)

Often involves the word “never”, “at most”, “cannot”,...

Sometimes called “partial correctness”

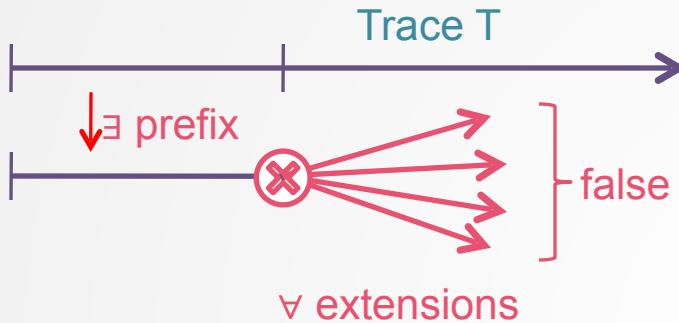
MORE ON LIVENESS

Liveness can only be
satisfied in finite time (when the good happens)
violated in infinite time (there's always hope)

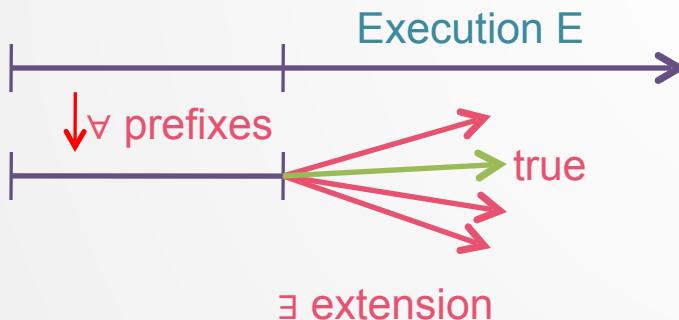
Often involves the words **eventually**, or must
Eventually means at some (often unknown)
point in “future”

Liveness is often just “termination”

FORMAL DEFINITIONS VISUALLY



- Safety is violated for an execution E if there exists a prefix such that **all** extensions are false



- Liveness is satisfied (true) for an execution E if for all prefixes there exists an extension that is true

PONDERING SAFETY AND LIVENESS

Is really every property either liveness or safety?

“Every message should be delivered exactly once”

Every message is delivered at most once **and**

Every message is delivered at least once

Process Failure Model

SPECIFICATION OF A SERVICE

How to specify a distributed service (abstract)?

Interface (aka Contract, API)

Requests

Responses

Correctness Properties

Safety

Liveness

Model

Assumptions on failures

Assumptions on timing (amount of synchrony)

Implementation

Composed of other services

Adheres to interface and satisfies correctness

Has internal events

declarative specification

“what”

aka problem

imperative,
many possible
“how”

MODEL/ASSUMPTIONS

Specification needs to specify the distributed computing model

- Assumptions needed for the algorithm to be correct

Model includes **assumptions** on

- Failure behavior of processes & channels
- Timing behavior of processes & channel

PROCESS FAILURES

Processes may fail in four ways:

- Crash-stop
- Omissions
- Crash-recovery
- Byzantine/Arbitrary
- Processes that don't fail in an execution are **correct**

CRASH-STOP FAILURES

- Crash-stop failure
 - Process stops taking steps
 - Not sending messages
 - Nor receiving messages
 - Default failure model is crash-stop
 - Hence, do not recover
 - But are processes not allowed to recover?

OMISSION FAILURES

- Process omits sending or receiving messages
 - Some differentiate between
 - **Send omission**
 - Not sending messages the process has to send according to its algorithm
 - **Receive omission**
 - Not receiving messages that have been sent to the process
 - For us, omission failure covers both types

CRASH-RECOVERY FAILURES

The process might crash

It stops taking steps, not receiving and sending messages

It may **recover** after crashing

Special **<Recovery>** event automatically generated

Restarting in some **initial recovery state**

Has access to **not-volatile storage**

May read/write (**expensive**) to permanent storage device

Storage survives crashes

E.g., save state to storage, crash, recover, read saved state

CRASH-RECOVERY FAILURES

- Failure is different in crash-recovery model
 - A process is **faulty** in an execution if
 - It crashes and **never** recovers, or
 - It crashes and recovers **infinitely often** (**unstable**)
 - Hence, a **correct process** may crash and recover
 - As long as it is a finite number of times

BYZANTINE FAILURES

- Byzantine/Arbitrary failures
 - A process may behave arbitrarily
 - Sending messages not specified by its algorithm
 - Updating its state as not specified by its algorithm
 - May behave **maliciously**, attacking the system
 - Several malicious processes might collude

Fault-tolerance Hierarchy

FAULT-TOLERANCE HIERARCHY

- Is there a hierarchy among the failure types
 - Which one is a special case of which?
 - An algorithm that works correctly under a general form of failure, works correctly under a special form of failure
- Crash special case of Omission
 - Omission restricted to omitting everything after a certain event

FAULT-TOLERANCE HIERARCHY

- In Crash-recovery
 - Under assumption that processes use stable storage as their main memory
- Crash-recovery is identical to omission
 - Crashing, recovering, and reading last state from storage
 - Just same as omitting send/receiving while being crashed

FAULT-TOLERANCE HIERARCHY

- In crash-recovery it is possible to use volatile memory
 - Then recovered nodes might not be able to restore all of state
 - Thus crash-recovery extends omission with **amnesia**
- Omission is special case of Crash-recovery
 - Crash-recovery , not allowing for amnesia

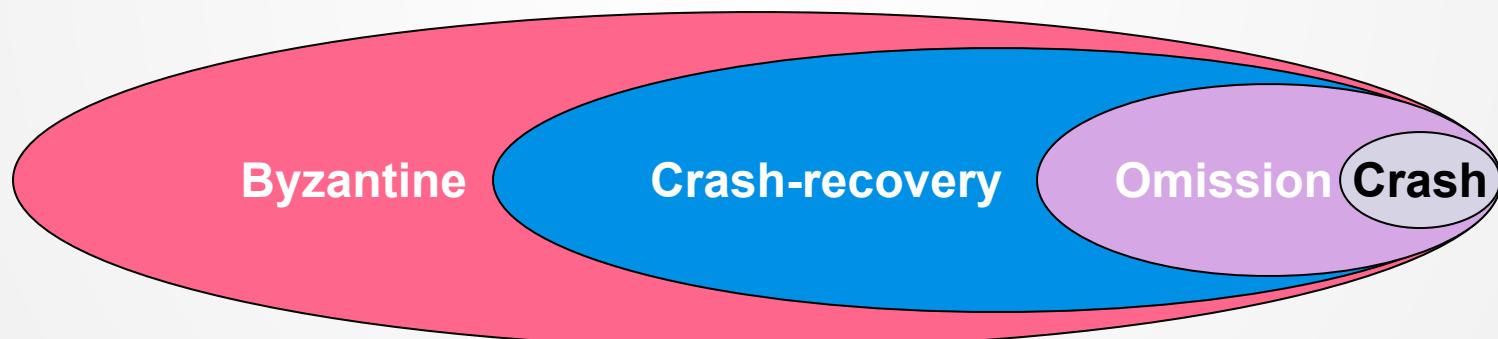
FAULT-TOLERANCE HIERARCHY

Crash-recovery special case of Byzantine

Since Byzantine allows anything

Byzantine tolerance → crash-recovery tolerance

Crash-recovery → omission, omission → crash-stop



Channel Behavior (failures)

SPECIFICATION OF A SERVICE

How to specify a distributed service (abstract)?

Interface (aka Contract, API)

Requests

Responses

Correctness Properties

Safety

Liveness

Model

Assumptions on failures

Assumptions on timing (amount of synchrony)

Implementation

Composed of other services

Adheres to interface and satisfies correctness

Has internal events

declarative specification

“what”

aka problem

imperative,
many possible
“how”

CHANNEL FAILURE MODES

- Fair-Loss Links

- Channels delivers any message sent with non-zero probability (no network partitions)

- Stubborn Links

- Channels delivers any message sent infinitely many times

- Perfect Links

- Channels that delivers any message sent exactly once

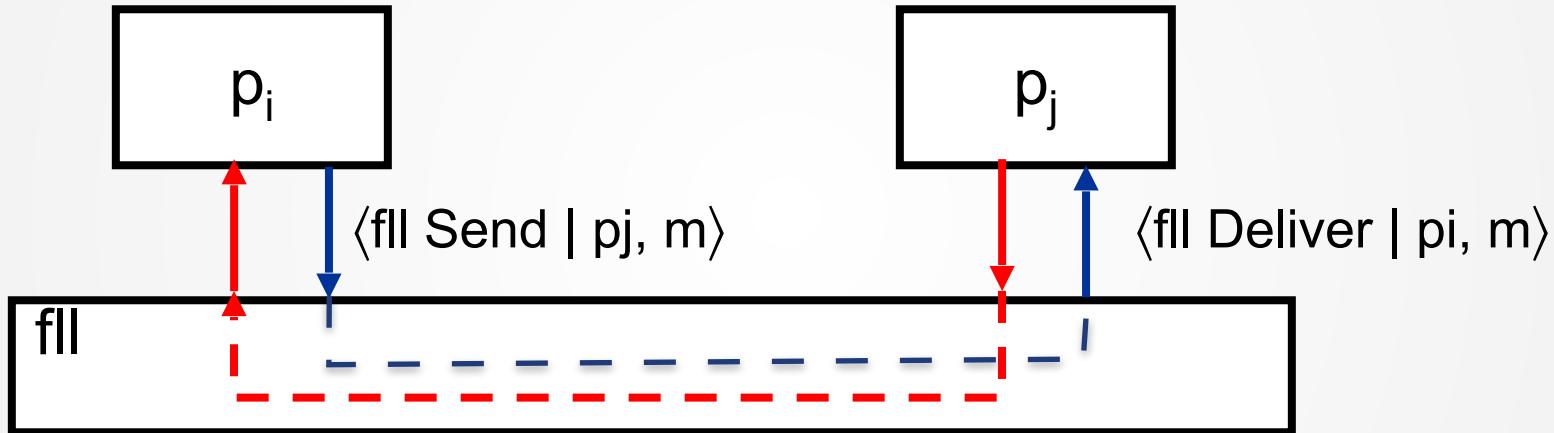
Fair-Loss (Hopeful) Links

CHANNEL FAILURE MODES

Fair-Loss Links

Channels delivers any message sent with non-zero probability (no network partitions)

FAIR LOSS LINKS (FLL)





FAIR-LOSS LINKS: INTERFACES

Module:

Name: FairLossPointToPointLink **instance** fll

Events:

Request: <fll, Send | dest, m>

Request transmission of message m to process dest

Indication:<fll, Deliver | src, m>

Deliver message m sent by process src

Properties:

FL1, FL2, FL3.



FAIR-LOSS LINKS

Properties

FL1. Fair-loss: If m is sent infinitely often by p_i to p_j , and neither crash, then m is delivered infinitely often by p_j

FL2. Finite duplication: If a m is sent a finite number of times by p_i to p_j , then it is delivered at most a finite number of times by p_j

I.e. a message cannot be duplicated infinitely many times

FL3. No creation: No message is delivered unless it was sent

Stubborn Links

CHANNEL FAILURE MODES

Stubborn Links

Channels delivers any message sent infinitely many times



STUBBORN LINKS: INTERFACE

Module:

Name: StubbornPointToPointLink **instance** sl

Events:

Request: <sl, Send | dest, m>

Request the transmission of message m to process dest

Indication:<sl, Deliver src, m>

deliver message m sent by process src

Properties:

SL1, SL2



STUBBORN LINKS

- Properties
 - ***SL1. Stubborn delivery:*** if a correct process p_i sends a message m to a correct process p_j , then p_j delivers m an infinite number of times
 - ***SL2. No creation:*** if a message m is delivered by some process p_j , then m was previously sent by some process p_i



IMPLEMENTING STUBBORN LINKS

- Implementation
 - Use the Lossy (fair-loss) link
 - Sender stores every message it sends in **sent**
 - It periodically resends all messages in **sent**



ALGORITHM (SL)

Implements: StubbornLinks **instance** sl

Uses: FairLossLinks, **instance** fll

- **upon event** <sl, Init> **do**

sent := \emptyset

startTimer(TimeDelay)

- **upon event** <sl, Send | dest, m> **do**

trigger <fl, Send | src, m>

sent := sent $\cup \{ (dest, m) \}$

- **upon event** <fl, Deliver | src, m> **do**

trigger <sl Deliver | src, m>

- **upon event** <Timeout> **do**

forall (dest, m) \in sent **do**

trigger <fl, Send | dest, m>

startTimer(TimeDelay)



IMPLEMENTING STUBBORN LINKS

- Implementation

- Use the Lossy link
- Sender stores every message it sends in **sent**
 - It periodically resends all messages in **sent**

- Correctness

- ***SL1. Stubborn delivery***

- If process doesn't crash, it will send every message infinitely many times. Messages will be delivered infinitely many times. Lossy link may only drop a (large) fraction.

- ***SL2. No creation***

- Guaranteed by the Fair-Lossy link

Perfect Links

CHANNEL FAILURE MODES

- Perfect Links

- Channels that deliver any message sent exactly once



PERFECT LINKS: INTERFACE

- **Module:**

- Name: PerfectPointToPointLink, **instance** pl

- **Events:**

- **Request:** $\langle pl, \text{Send} \mid dest, m \rangle$

- Request the transmission of message m to node dest

- **Indication:** $\langle pl, \text{Deliver} \mid src, m \rangle$

- deliver message m sent by node src

- **Properties:**

- **PL1, PL2, PL3**



PERFECT LINKS (RELIABLE LINKS)

Properties

- ***PL1. Reliable Delivery (at least once):***
If p_i and p_j are correct, then every message sent by p_i to p_j is eventually delivered by p_j
- ***PL2. No duplication:*** Every message is delivered at most once
- ***PL3. No creation:*** No message is delivered unless it was sent



PERFECT LINKS (RELIABLE LINKS)

Which one is safety/liveness/neither

- (liveness) **PL1. Reliable Delivery:** If neither p_i nor p_j crashes, then every message sent by p_i to p_j is eventually delivered by p_j
- (safety) **PL2. No duplication:** Every message is delivered at most once
- (safety) **PL3. No creation:** No message is delivered unless it was sent



PERFECT LINK IMPLEMENTATION

- Implementation
 - Use Stubborn links
 - Receiver keeps a **log** of all received messages in **Delivered**
 - Only deliver (perfect link Deliver) messages that weren't delivered before
- Correctness
 - *PL1. Reliable Delivery*
 - Guaranteed by Stubborn link. In fact the Stubborn link will deliver it infinite number of times
 - *PL2. No duplication*
 - Guaranteed by our log mechanism
 - *PL3. No creation*
 - Guaranteed by Stubborn link (and its lossy link?)



FIFO PERFECT LINKS (RELIABLE LINKS)

Properties

PL1. Reliable Delivery:

PL2. No duplication:

PL3. No creation: No message is delivered unless it was sent

FFPL. Ordered Delivery: if m_1 is sent before m_2 by p_i to p_j and m_2 is delivered by p_j then m_1 is delivered by p_j before m_2

INTERNET TCP VS. FIFO PERFECT LINKS

- TCP provides reliable delivery of packets
- TCP reliability is so called “session based”
- Uses sequence numbers
 - ACK: “I have received everything up to byte X”
- Implementing Perfect Link abstraction on TCP requires reconciling messages between the sender and receiver when reestablishing connection after a session break

DEFAULT ASSUMPTIONS IN COURSE

- We **assume perfect links** (aka reliable) most of time in the course (unless specified otherwise)
- Roughly, reliable links ensure messages exchanged between correct processes are delivered exactly once
- Messages are **uniquely** identified and
 - the message identifier includes the sender's identifier
 - i.e. if “same” message sent twice, it’s considered as two different messages
- Many algorithm for crash-recovery process model assume either a Stubborn link, or (Logged) perfect link

Timing Assumptions

SPECIFICATION OF A SERVICE

How to specify a distributed service (abstract)?

Interface (aka Contract, API)

Requests

Responses

Correctness Properties

Safety

Live ness

Model

Assumptions on failures

Assumptions on timing (amount of synchrony)

Implementation

Composed of other services

Adheres to interface and satisfies correctness

Has internal events



declarative specification
“what”
aka problem

imperative,
many possible
“how”

TIMING ASSUMPTIONS

- Timing assumptions
 - Processes
 - bounds on time to make a computation step
 - Network
 - Bounds on time to transmit a message between a sender and a receiver
 - Clocks:
 - Lower and upper bounds on clock rate-drift and clock skew w.r.t. real time

RECAP - MODELS

- Synchronous (systems build on solid timed operations + clocks)
- Partially Synchronous (eventually every execution will exhibit period of synchrony - to make progress - satisfy liveness)
- Asynchronous (?)

Causality in the Asynchronous Model

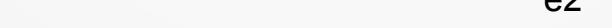
ASYNCHRONOUS SYSTEMS

- No timing assumption on processes and channels
 - Processing time is uncertain.
 - Network latency is uncertain.
 - No hardware clocks (that work) exist.
- Reasoning based only on which events may cause other events
 - We call this “Causality”
- Total order of events not observable locally, no access to global clocks

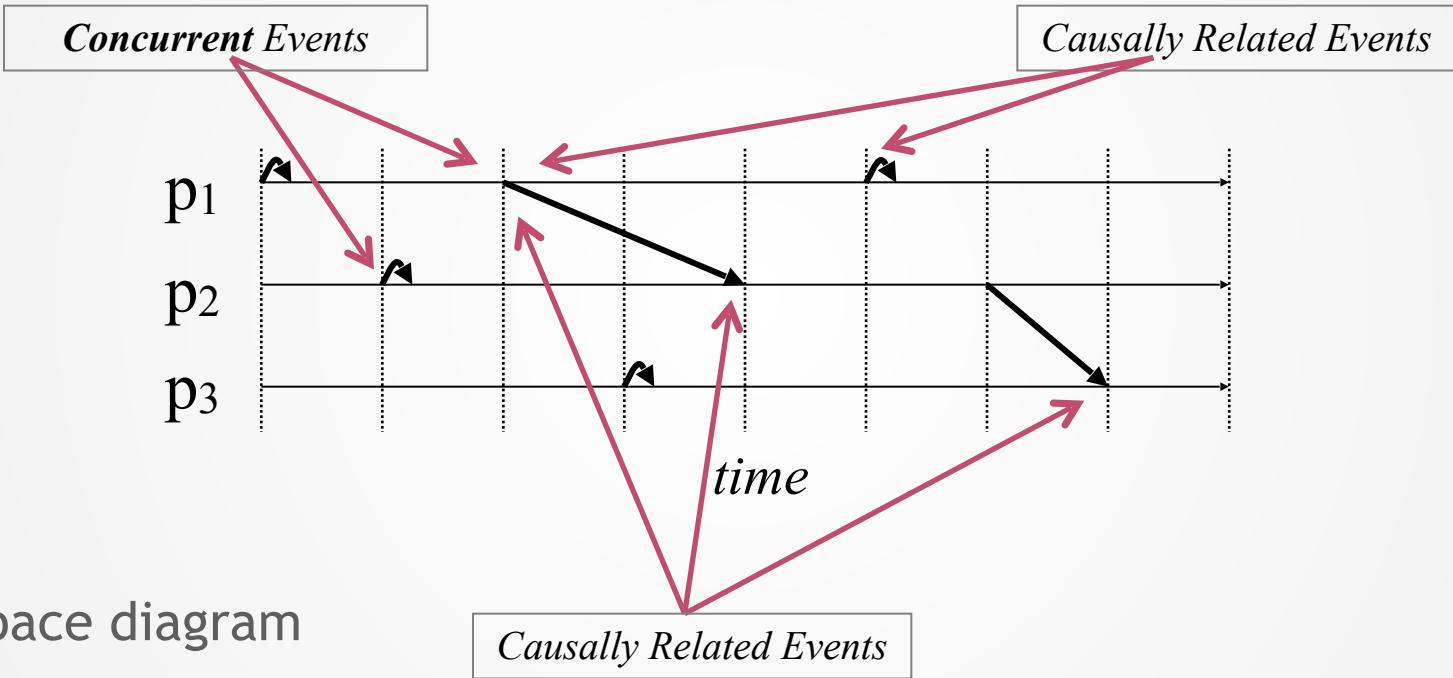
CAUSAL ORDER (HAPPEN BEFORE)

- Given an execution trace β ,
two events $a, b \in \beta$ are **causally ordered** ($a \rightarrow_{\beta} b$) iff either:
 - a occurs before b on the same process
 - a is a send(m) and b deliver(m) event
 - there exists a sequence of causally ordered events from a to b (**transitive**) e.g. If $a \rightarrow_{\beta} c$ and $c \rightarrow_{\beta} b$
- Two events, a and b , are **concurrent** if **not** $a \rightarrow_{\beta} b$ and **not** $b \rightarrow_{\beta} a$
 - Concurrent events are denoted as such: $a || b$

CAUSAL ORDER (HAPPEN BEFORE)



EXAMPLE



SIMILARITY OF EXECUTIONS

- The **view of p_i** in E , denoted $E|p_i$, is the subsequence of execution E restricted to events and state of p_i
- Two executions E and F are similar w.r.t p_i
if $E|p_i = F|p_i$
- Two executions E and F are similar
if $E|p_i = F|p_i$, $\forall p_i \in \Pi$
(Π is the set of all processes)

THE COMPUTATION THEOREM

- Let E be an execution $(c_0, e_1, c_1, e_2, c_2, \dots)$, and V the trace of events (e_1, e_2, e_3, \dots)
- Let P be a permutation of V , preserving causal order
 - $P = (f_1, f_2, f_3, \dots)$ preserves the causal order of V when for every pair of events $f_i \rightarrow_V f_j$ implies $f_i \rightarrow_P f_j$
- E is **similar** to the execution starting in c_0 with trace P

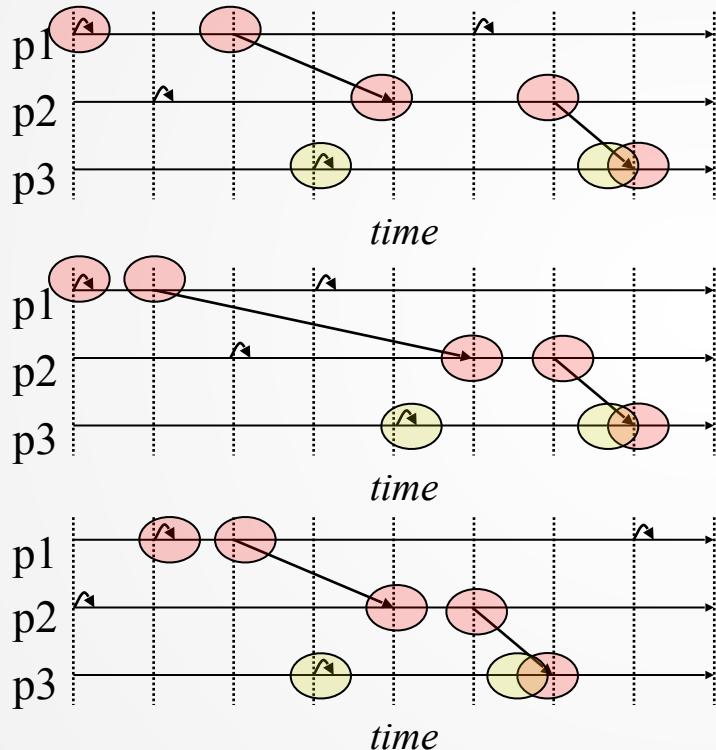
EXECUTION EQUIVALENCE

- If two executions F and E have the same collection of events, and their causal order is preserved, F and E are said to be **similar executions**, written $F \sim E$
 - F and E could have different permutation of events as long as causality is preserved!

COMPUTATIONS

- Similar executions form **equivalence classes** where every execution in a class is similar to the other executions in the same class
- I.e. the following always holds for executions:
 - \sim is reflexive
 - I.e. $a \sim a$ for any execution
 - \sim is symmetric
 - I.e. If $a \sim b$ then $b \sim a$ for any executions a and b
 - \sim is transitive
 - If $a \sim b$ and $b \sim c$, then $a \sim c$, for any executions a, b, c
- Equivalence classes are called **computations** of executions

EXAMPLE OF SIMILAR EXECUTIONS



● ● *Same color ~ Causally related*

All three executions are part
of the same computation, as
causality is preserved

A COMPUTATIONAL PARADOX

The computation theorem yields two “interesting” observations

Observation 1: There is no algorithm in the asynchronous system model that can observe the **global order** of the sequence of events (that can “see” the time-space diagram, or the trace) for all executions

Observation 2: The computation theorem does not hold if the model is extended such that each process can read a local **hardware clock**

A COMPUTATIONAL PARADOX

Proof of 1.

- Assume such an algorithm exists. Assume p knows the order in the final (repeated) configuration
- Take two distinct similar executions of algorithm preserving causality
- According to the computation theorem, their final repeated configurations are the **same**, then the algorithm cannot have observed the actual order of events as **they differ**.

A COMPUTATIONAL PARADOX

Proof of 2. :

- Similarly, assume a distributed algorithm in which each process reads the local clock each time a local event occurs
- The final (repeated) configuration of different causality preserving executions will have different clock values, which would contradict the computation theorem