Algorithms and Distributed Systems 2019/2020 (Lecture One)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Leitão (jc.leitao@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Lecture structure:

- Understand what is a distributed system.
- Modeling a distributed system.
  - General system model.
  - Fault model.
  - Timing assumption.
- Understanding the role of an algorithm.
  - Point-to-Point Communication Algorithms.
- The Broadcast Problem
  - Best effort broadcast.
  - Reliable broadcast.

What is a distributed system?

“A distributed system is a network that consists of autonomous computers that are connected using a distribution middleware. They help in sharing different resources and capabilities to provide users with a single and integrated coherent network.” – Technopedia.

What is a distributed system?

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

-- L. Lamport (Computer Science Turing Award)

What is a distributed system?

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

-- L. Lamport (Computer Science Turing Award)

This might be a more honest answer!

What is a distributed system?

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

-- L. Lamport (Computer Science Turing Award)

Notice that our goal, as designers of distributed systems and distributed algorithms is to avoid this definition to be correct!

This might be a more honest answer!

What is a distributed system?

A more honest and focused answer:

A distributed system is composed by a set of **processes** that are interconnected through some **network** where processes seek to achieve some form of **cooperation** to execute tasks.

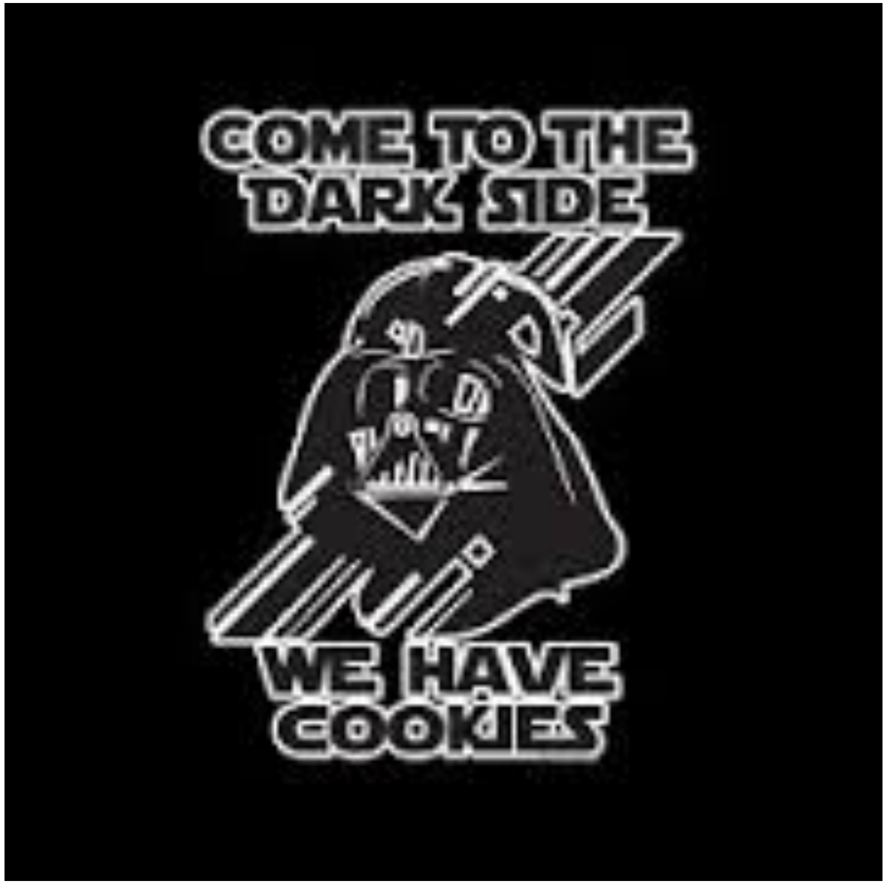
Why do we want to distribute things anyways?

Why do we want to distribute things anyways?

- **Fault-Tolerance (also known as *dependability*):** If I use N machines to support my system and **f** ($f < N$) fail, then my system can still operate.

- **Concurrency (or more processing/storage power)**: If instead of using 1 machine to run my system, I use N machines ($N >> 1$) then I will have N times more resources and hopefully my system will be (close to) N times faster.

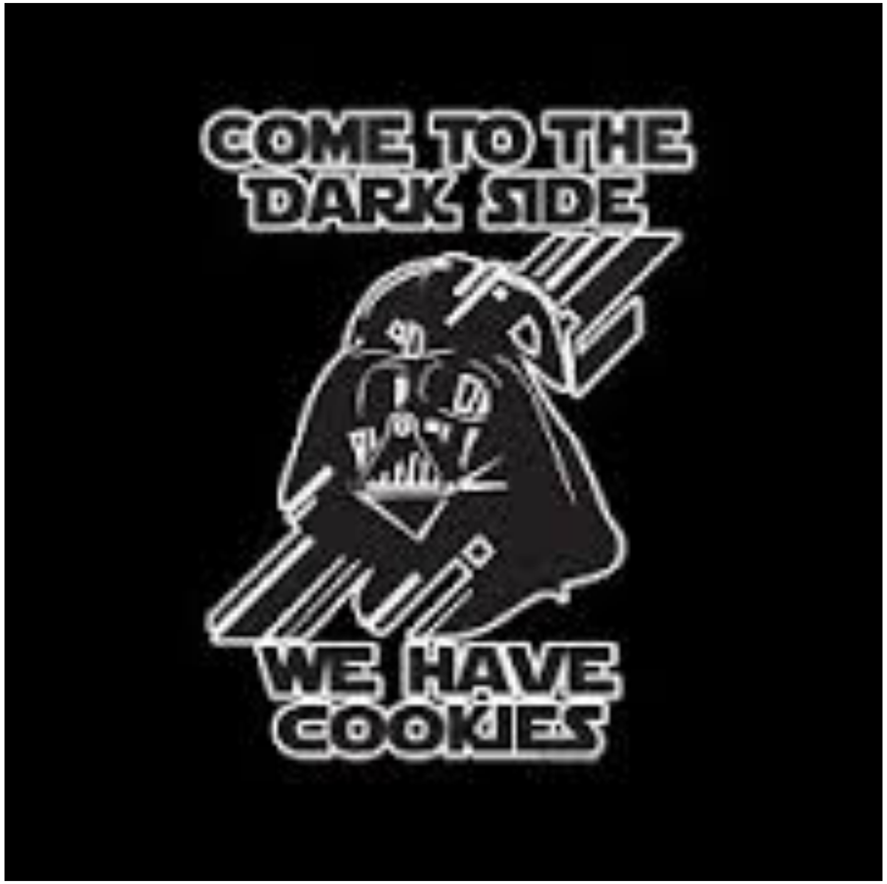
The "Dark Side" of Distributed Systems.



The “Dark Side” of Distributed Systems.

There are challenges inherent to distribution:

- **Fault-Tolerance:** It is necessary to consider what you do when a machine fails (and they will).
- **Concurrency:** It is necessary to consider the possible orderings of events.



Reasoning about Distributed Systems.

- By this time you can guess that distributed systems are complex.

Reasoning about Distributed Systems.

- By this time you can guess that distributed systems are complex.
- And our task is to understand how they behave and how we can build them to operate correctly (and efficiently).

Reasoning about Distributed Systems.

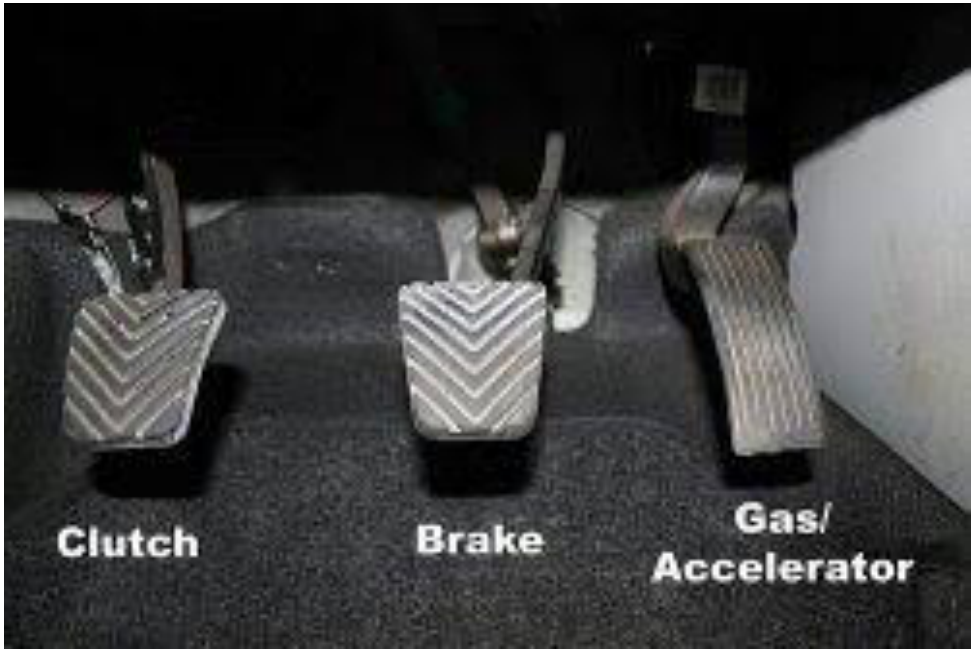
- By this time you can guess that distributed systems are complex.
- And our task is to understand how they behave and how we can build them to operate correctly (and efficiently).
- Oh oh...

The model of a distributed system

- Somewhat “less inteligent” example:
  - How do you reason about a car?

The model of a distributed system

- Somewhat “less inteligent” example:
  - How do you reason about a car?

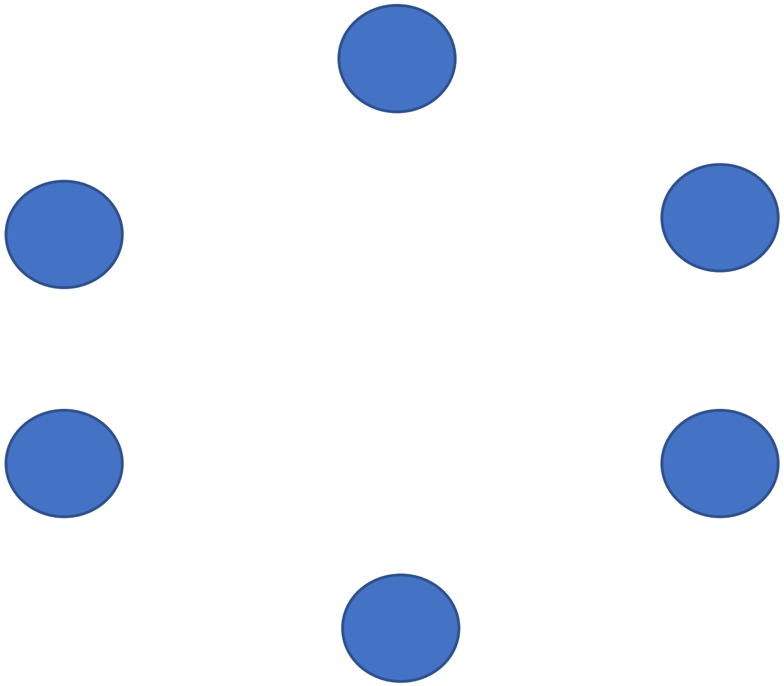


The model of a distributed system

A distributed system is composed by a set of processes that are interconnected through some network where processes seek to achieve some form of cooperation to execute tasks.

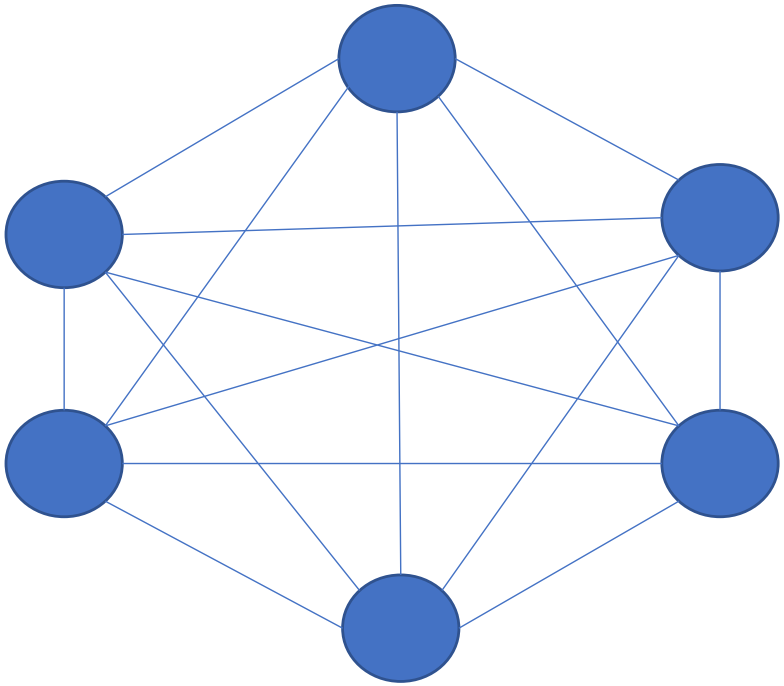
The model of a distributed system

A distributed system is composed by a set of **processes** that are interconnected through some network where processes seek to achieve some form of cooperation to execute tasks.



The model of a distributed system

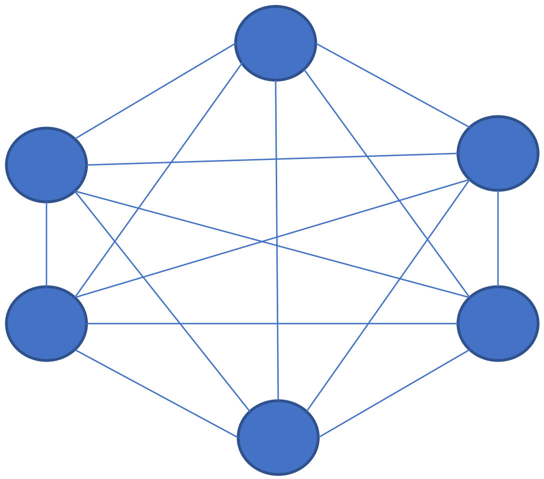
A distributed system is composed by a set of **processes** that are interconnected through some **network** where processes seek to achieve some form of cooperation to execute tasks.



The model of a distributed system

More formally:

- Processes – computational elements:
    - Abstracts the notion of machine/node.
- Network – Graph **G=(V,E)**, in which **V** is the set of processes, **E** represents the communication channels (i.e, links) between pairs of processes.

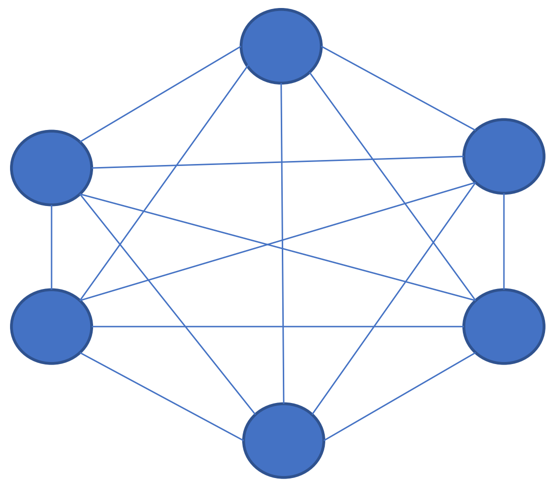


The model of a distributed system

More formally:

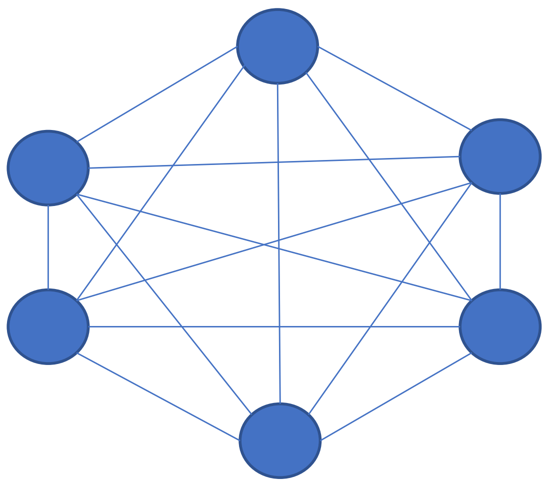
- Processes – computational elements:
  - Abstracts the notion of machine/node.
- Network – Graph **G=(V,E)**, in which **V** is the set of processes, **E** represents the communication channels (i.e., links) between pairs of processes.

In general, we will consider a complete graph, where every process is connected to every other by a bidirectional link.



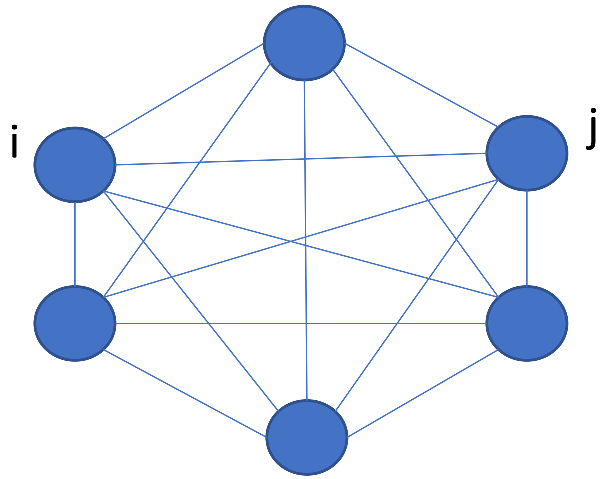
The model of a distributed system

- Processes are fully independent, meaning that they do not share memory in any way.
- But a clear necessity for cooperation is to exchange information… so how do processes exchange information among them?



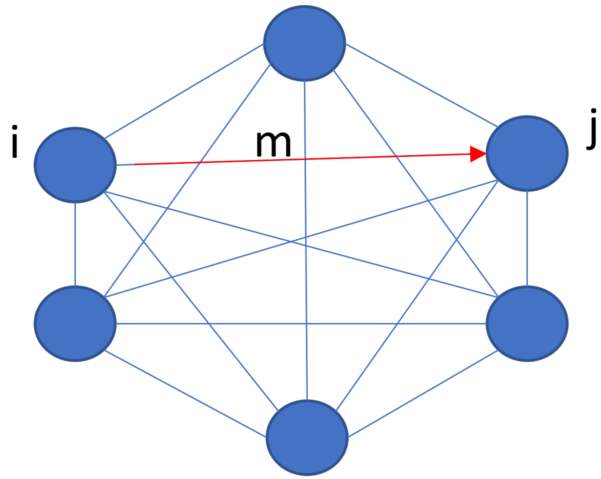
The model of a distributed system

- Processes are fully independent, meaning that they do not share memory in any way.
- But a clear necessity for cooperation is to exchange information… so how do processes exchange information among them?



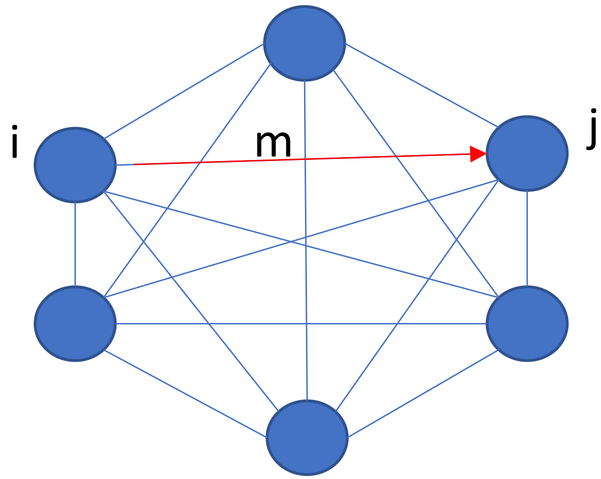
The model of a distributed system

- Processes are fully independent, meaning that they do not share memory in any way.
- But a clear necessity for cooperation is to exchange information… so how do processes exchange information among them?



The model of a distributed system

- More formally:
- Processes communicate through the exchange of Messages, belonging to an alphabet **M** plus the special symbol **null**, which captures a non-existent message.
- Notation: $send_i(j, m, arg_1, arg_2, \ldots, arg_n)$
  - Process **i** send message **m** with arguments arg_1 to arg_n to process **j**.

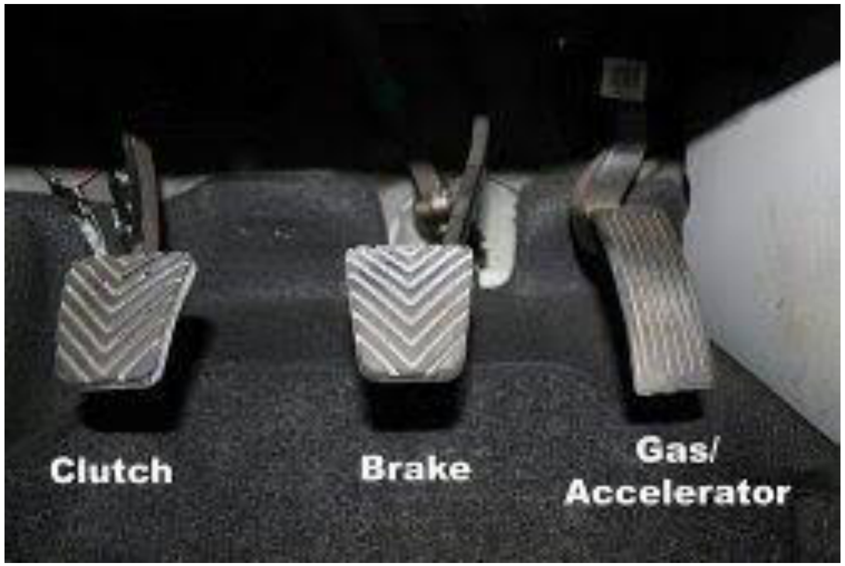


The model of a distributed system

- So we are done right?

The model of a distributed system

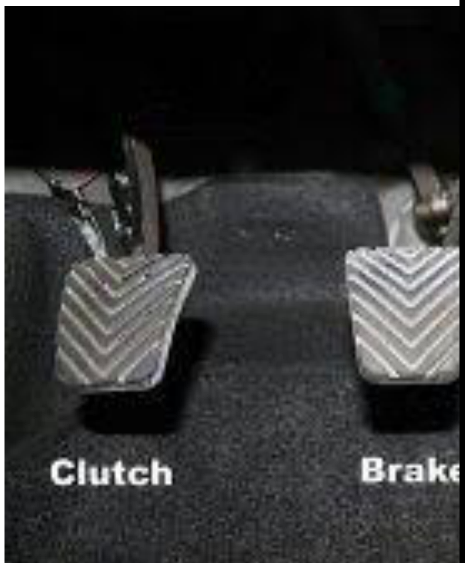
- So we are done right?
- Not quite, back to our somewhat “less inteligent” example:
  - How do you reason about a car?




tributed system

ewhat “less inteligent”

How do yo



The model of a distributed system

- So we are done right?
- Not quite...
- Some extra things have to be considered.
  - Timing assumptions.
  - Internal model of the process.
  - (Process) Fault model.
  - Network model,

Timing assumptions

- They capture what is assumed in relation to time in the system:
- Two fundamental models:
- Synchronous System:
  - One assumes that there is a known upper bound to the time required to deliver a message through the network and for a process to make all computations related with the processing of the message.
- Asynchronous System:
  - The are no assumptions about the time require to deliver a message or process a message.

Timing assumptions

- This might look as not a big deal but actually there are strong implications here.
  - In a synchronous system you can detect when a process fails (in some particular faults models).
  - In a synchronous system you can have protocols evolve in synchronous steps (Why is that?)
  - In an asynchronous system there are some problems that actually have no solution.

Timing assumptions

- This might look as not a big deal but actually there are strong implications here.
  - In a synchronous system you can detect when a process fails (in some particular faults models).
  - In a synchronous system you can have protocols evolve in synchronous steps (Why is that?)
  - In an asynchronous system there are some problems that actually have no solution.
  - **The “real world” is actually asynchronous**, so why is it that we sometimes consider the synchronous model?

Beyond Synchrony and Asynchrony

- There is a practical system model that in some sense stands between these two extremes.
- The partially synchronous model (or eventually synchronous model).
- In this model the system is considered to be **asynchronous** but it is assumed that **eventually** (*meaning for sure at some time in the future that is unknown*) the system will behave in a **synchronous** way for long enough.

Internal Model of the Process

- Each process has a unique identifier: $\mathbf{i} \in \mathbf{V}$
- Internally each process has (classical model):
  - $states_i$ – set of states
  - $start_i$ – initial state(s)
  - Inputs and outputs are special state variables (allow the process to get information from outside and export information to the outside)
  - $init_i$ – intial state.
  - Fundamentally, a process is a **deterministic automaton**.

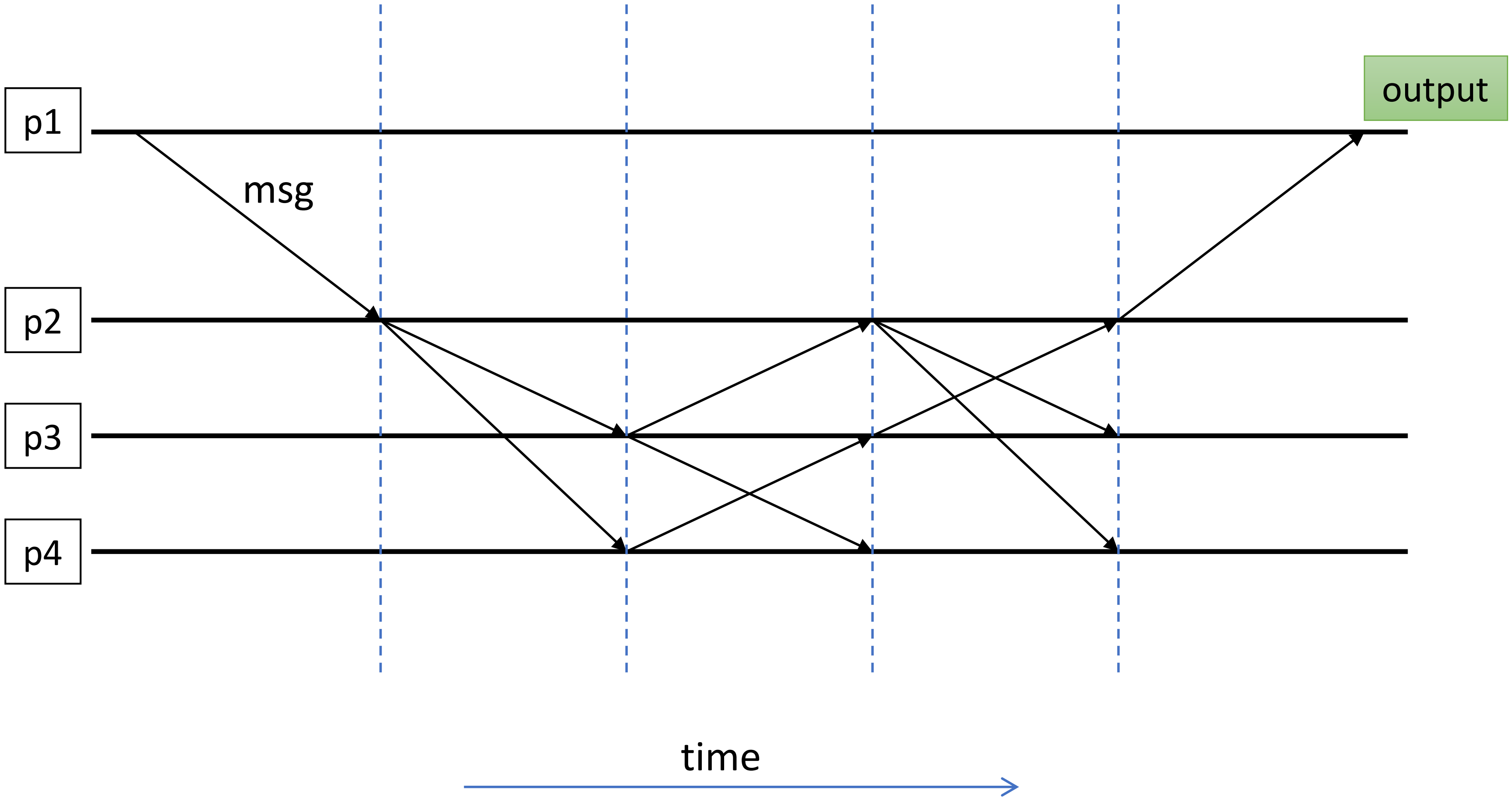
Transition between States

- Synchronous Model: Execution in rounds.
- Transition of the process state based on two functions:
  - $trans_i$: $states_i \times [M \cup \{null\}]^V \rightarrow states_i$
  - $msgs_i$: $states_i \rightarrow [M \cup \{null\}]^V$
- In each round a process will:
  - Receive messages from all processes.
  - Process messages to determine which messages are generated.
  - Send messages to all processes.
  - Apply $trans_i$ to determine its following state.

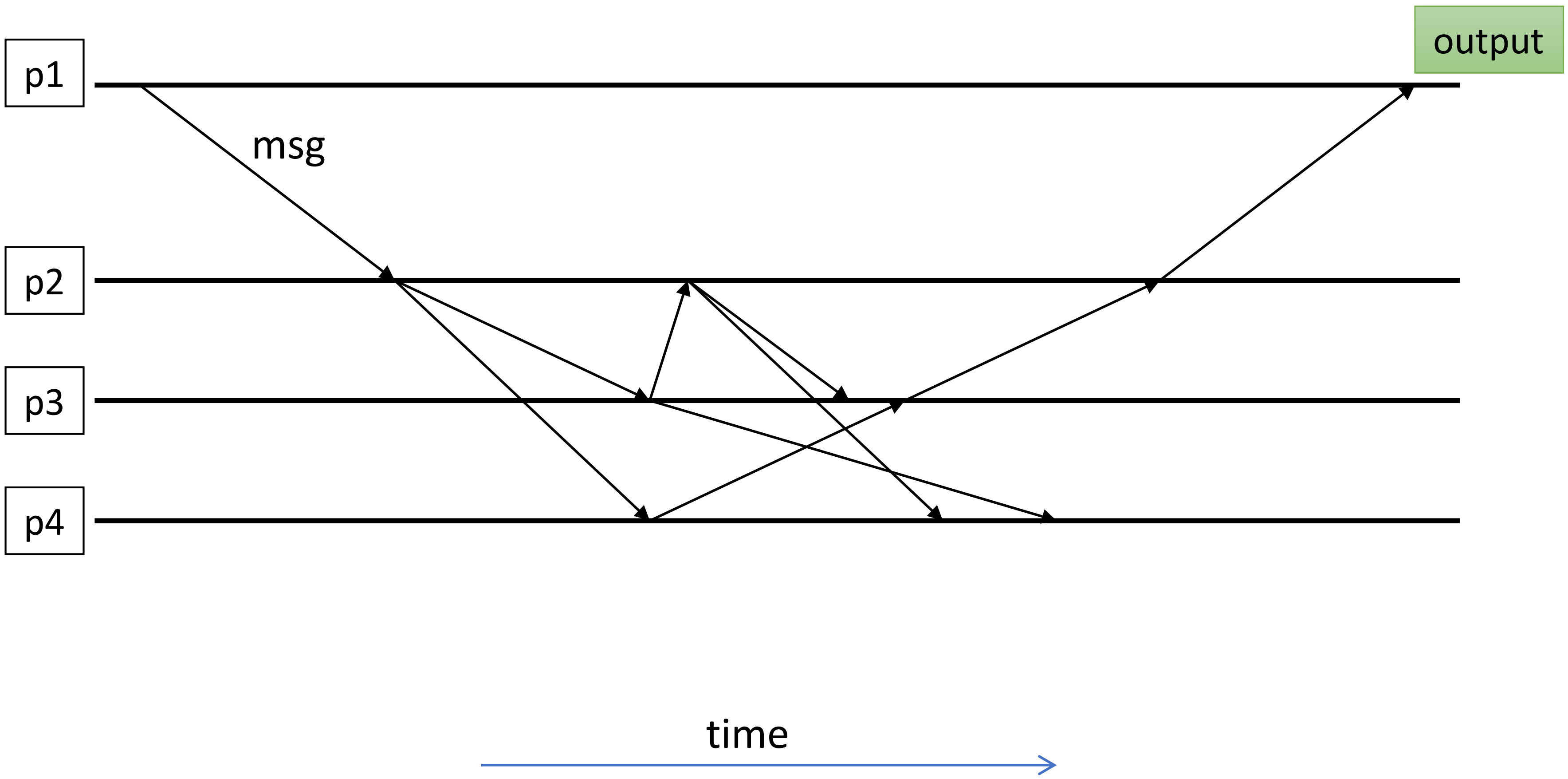
Transition between States

- Asynchronous Model: Execution is not based on rounds.
- Transition of the process state based on two functions:
  - $trans_i$: $states_i \times M \cup \{null\} \rightarrow states_i$
  - $msgs_i$: $states_i \rightarrow \{M \cup \{null\}\}^V$
- Since there is no notion of rounds:
  - A transition of state is triggered by the reception of a single message (notice that the transition can be S → S)
  - Transitioning to a state (even if the same) can trigger the generation (and transmission) of a new set of messages.

Example of an execution in the synchronous model:

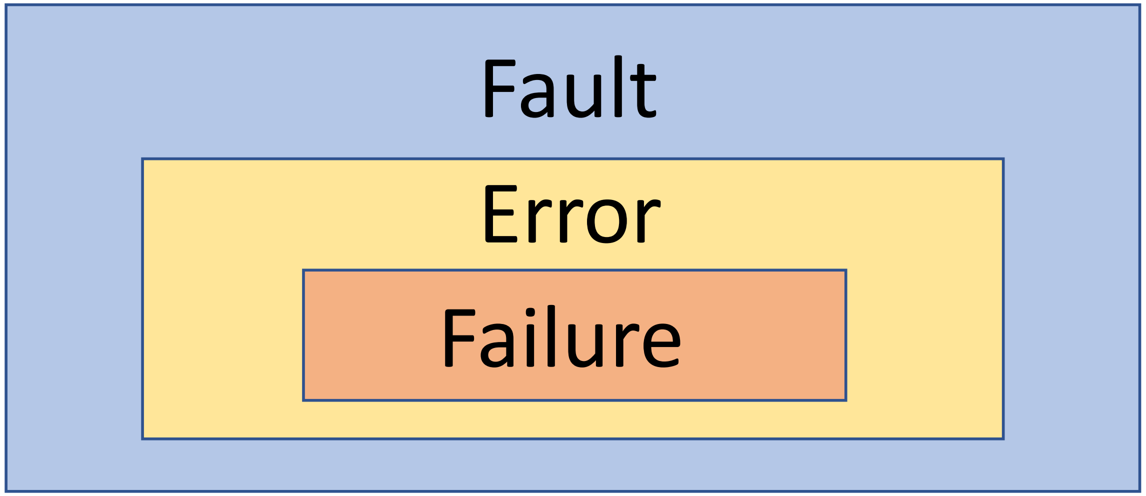


Example of an execution in the asynchronous model:



Fault Model

- Faults lead processes to deviate from their expected behaviour.
- Classical Model:



- Example: Sector in the hard disk is damaged (fault); -> Sector is Accessed (Error) -> File is lost (Failure)

Fault Model (cont.)

- This classical model usually has a recursive implication.
- The failure of a component of a process might imply a fault in another (higher level component).
- Going back to the previous example, the failure of the file system (file damaged) might lead to a fault in the load of the operative system, which might result in the failure of the operative system.

Process Fault Model

- A process that never fails, is considered **correct.**
- **Correct processes** never deviate from their expected/prescribed behaviour.
  - It executes the algorithm as expected and sends all messages prescribed by it.
- **Failed processes** might deviate from their prescribed behaviour in diferent ways.
  - The *unit of failure* is the process, meaning that when it fails, all its component fail at the same time.
- The (possible) behaviours of a process that fails is defined by the process fault model.

Process Fault Models

- Crash Fault Model:
  - When a process fails it stops sending any messages (from that point onward).
  - This is the fault model that we will consider most of the times.

Process Fault Models

- Crash Fault Model:
  - When a process fails it stops sending any messages (from that point onward).
  - This is the fault model that we will consider most of the times.
- Omission Fault Model:
  - A process that fails omits the transmission (or reception) of any number of messages.
- Fail-Stop Model:
  - Similar to the crash model, except that upon failure the process "notifies" all other processes of its own failure.

Process Fault Models

- Byzantine (or Arbitrary) Fault Model:
  - A failed process might deviate from its protocol in any ways.
  - Examples:
    - Duplicate Messages,
    - Create invalid messages,
    - Modify values received from other processes,
  - Why is this relevant?

Process Fault Models

- Byzantine (or Arbitrary) Fault Model:
  - A failed process might deviate from its protocol in any ways.
  - Examples:
    - Duplicate Messages;
    - Create invalid messages;
    - Modify values received from other processes.
  - Why is this relevant?
    - Can capture memory corruption;
    - Can capture software bugs;
    - A malicious attacker that controls a process.

Process Fault Models

- Byzantine (or Arbitrary) Fault Model:
  - A failed process might deviate from its protocol in any ways.
  - Examples:
    - Duplicate Messages;
    - Create invalid messages;
    - Modify values received from other processes.
  - Why is this relevant?
    - Can capture memory corruption;
    - Can capture software bugs;
    - A malicious attacker that controls a process.
  - This is not something that we will delve here:
    - **Confiabilidade de Sistemas Distríbuidos deals with it.**

Network Model

- The Network Model captures the assumptions made concerning the **links** that interconnect processes.
- Namely it captures what can go wrong in the network regarding:
  - Lost of messages sent between processes.
  - Possibility of duplication of messages.
  - Possibility for corruption of messages.

Network Model

- The Network Model captures the assumptions made concerning the **links** that interconnect processes.
- Namely it captures what can go wrong in the network regarding:
  - Lost of messages sent between processes.
  - Possibility of duplication of messages.
  - Possibility for corruption of messages.

Network Model

- The Network Model
  made conc
  pr

The Network is usually not your friend.

Network Model (a starting point)

- Fair Loss Model
- A model that captures the possibility of messages being lost albeit in a fair way:
- Properties:
  - FL1 (Fair-Loss): Considering two correct processes ***i*** and ***j***; if ***i*** sends a message to ***j*** infinite times, then ***j*** delivers the message infinite times.
  - FL2 (Finite Duplication): Considering two correct processes ***i*** and ***j***; if ***i*** sends a message *m* to ***j*** a finite number of times, then ***j*** cannot deliver *m* infinite times.
  - FL3 (No Creation): If a correct process ***j*** delivers a message *m*, then *m* was sent to ***j*** by some process ***i*.**

Network Model (moving on…)

- Stubborn Model:
- A stronger model that assumes that processes communicate in a stubborn way:
- Properties:
  - SL1 (Stubborn Delivery): Considering two correct processes ***i*** and ***j***; if ***i*** sends a message to ***j***, then ***j*** delivers the message an infinite number of times.
  - SL2 (No Creation): If a correct process ***j*** delivers a message *m*, then *m* was sent to ***j*** by some process ***i.***

Network Model (better now)

- Perfect Link Model (also called Reliable):
- A stronger model that assumes the links between processes are well behaved:
- Properties:
  - PL1 (Reliable Delivery): Considering two correct processes ***i*** and ***j***; if ***i*** sends a message to ***j***, then ***j*** ***eventually*** delivers *m*.
  - PL2 (No Duplication): No message is delivered by a process more than once.
  - PL3 (No Creation): If a correct process ***j*** delivers a message *m*, then *m* was sent to ***j*** by some process ***i.***

How does all of this fits in?

- Our networks are actually closer to the fair-loss model, however its frequent that we use the perfect link model… Why?

How does all of this fits in?

- Our networks are actually closer to the fair-loss model, however its frequent that we use the perfect link model… Why?
  - The perfect link model makes it easier to reason about algorithms desing...

How does all of this fits in?

- Our networks are actually closer to the fair-loss model, however its frequent that we use the perfect link model… Why?
  - The perfect link model makes it easier to reason about algorithms desing...
  - ...but more importantly, these abstractions can be built on top of one another through the use of **distributed algorithms.**

How does all of this fits in?

- Our networks are actually closer to model, however its frequent that w perfect link model... Why?
  - The perfect link model makes it easier algorithms desing...
  - ...but more importantly, these abstrac on top of one another through the us **algorithms.**
- **What is this dark sorcery that you speak?**

How does all of this fits in?

- Our networks are actually closer to model, however its frequent that w perfect link model… Why?
  - The perfect link model makes it easier algorithms desing...
  - ...but more importantly, these abstrac on top of one another through the us **algorithms.**

- **What is this dark sorcery that you speak?**
- **Wait for the lab today to discover...**

How does all

Don’t forget that the perfect link abstraction does not states anything about time

•

• W

Algorithms Specification and Properties

- Noticed that when discussing these network models (i.e., abstractions) I have defined them in terms of a set of properties?

Algorithms Specification and Properties

- Noticed that when discussing these network models (i.e., abstractions) I have defined them in terms of a set of properties?
- Algorithms (that materialize these abstractions) also provide a set of properties (if correct, those of the abstraction they implement).
- Why do we tend to think in terms of properties?

Algorithms Specification and Properties

- Noticed that when discussing these network models (i.e, abstractions) I have defined them in terms of a set of properties?
- Algorithms (that materialize these abstractions) also provide a set of properties (if correct, those of the abstraction they implement).
- Why do we tend to think in terms of properties?
- Quick answer: Because algorithms are *composable*, and the design of an algorithm depends on the underlying properties provided by other algorithms.

Algorithms Specification and Properties

- What does these properties capture?
  - The correctness criteria for the algorithm (and for any implementation of that algorithm).
  - It defines restrictions to (all) valid executions of the algorithm.
- Two fundamental types of properties:
  - Safety
  - Liveness

Safety Properties

- Conditions that must be enforced at any (and all) point of the execution
  - Intuitively, bad things that should never happen.
- Relevant aspects:
  - The trace of an empty execution is always safe (do nothing and you shall do nothing wrong).
  - The prefix of a trace that does not violate safety, will never violate safety.

Liveness Properties

- Conditions that should be enforced at some point of an execution (but not necessarily always).
  - Intuitively, good things that should happen eventually (in the English sense of the word).
- Relevant aspects:
  - One can always extend the trace of an execution in a way that will respect liveness conditions (if you haven't done anything good yet, you might do it next).

Safety VS Liveness Properties

- Correct algorithms will have both Safety and Liveness properties.
- Some properties however are hard to classify within one of these classes, and they might mix aspects of safety and liveness
  - Usually one can decompose these properties in simpler ones through conjunctions.

We have now covered the main aspects of modelling distributed systems…

- A good time to give you a quote for you to remember:

“These are my principles. If you don’t like them, I have others” – Groucho Marx

Let’s now start to think about concrete problems…

- Now that we have covered the rules of the game, we can start to think on how to solve problems (with a particular set of rules).

The Broadcast Problem

- Informally: A process needs to transmit the same message *m* to N other processes.

- *Lets assume that the complete set of processes in the system is known a-priori:* **π**
- *Lets assume we have access to the Perfect Link Abstraction.*
- *Lets assume an asychronous system (no rounds, no failure detection).*

The Broadcast Problem

- Wait... How do you specify an algorithm again?

- I have sort of told you before, through a deterministic automaton...

The Broadcast Problem

- Wait... How do you specify an algorithm again?
- I have sort of told you before, through a deterministic automaton...
- Ok lets simplify this...

The Anatomy of an Algorithm:

ALGORITHM

The Anatomy of an Algorithm:

ALGORITHM

You need an Interface for this to be used (think of APIs)

The Anatomy of an Algorithm:

ALGORITHM

You need an Interface for this to be used (think of APIs)

Let’s use an event driven interface.

The Anatomy of an Algorithm:

Perfect (Point-to-Point) Link Algorithm

You need an Interface for this to be used (think of APIs)

Let's use an event driven interface.

The Anatomy of an Algorithm:

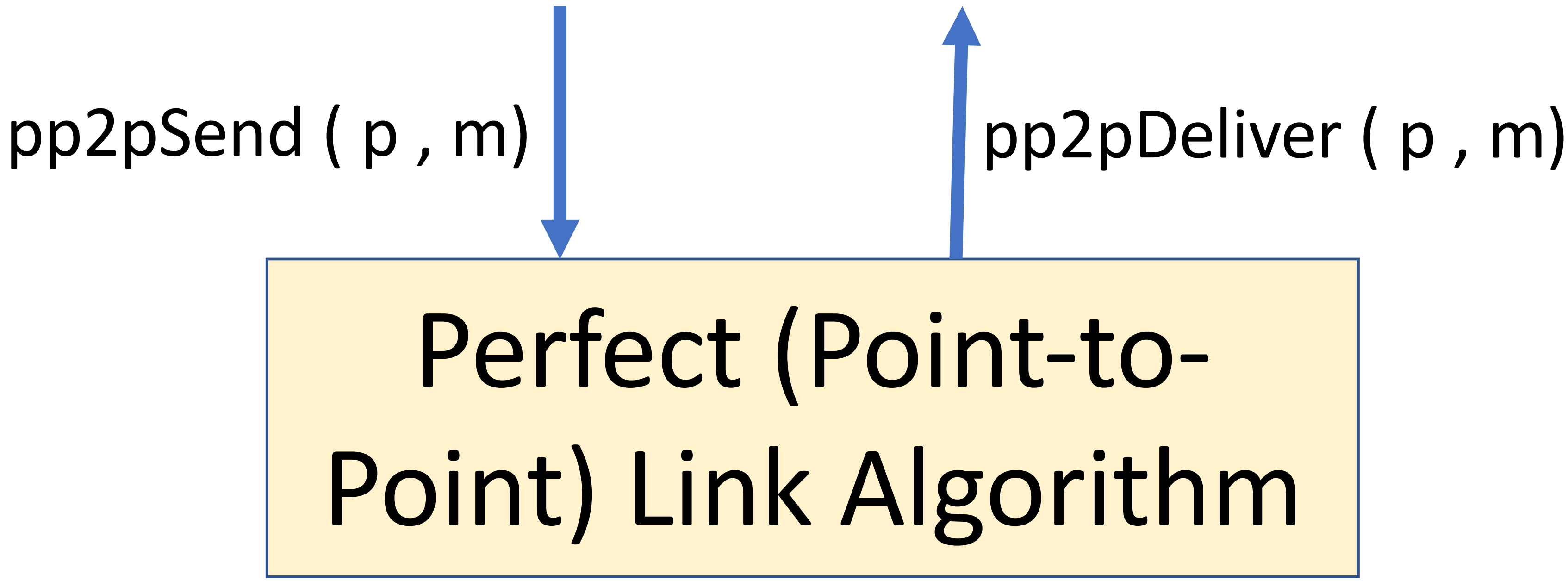
pp2pSend (p , m)

Perfect (Point-to-Point) Link Algorithm

You need an Interface for this to be used (think of APIs)

Let’s use an event driven interface.

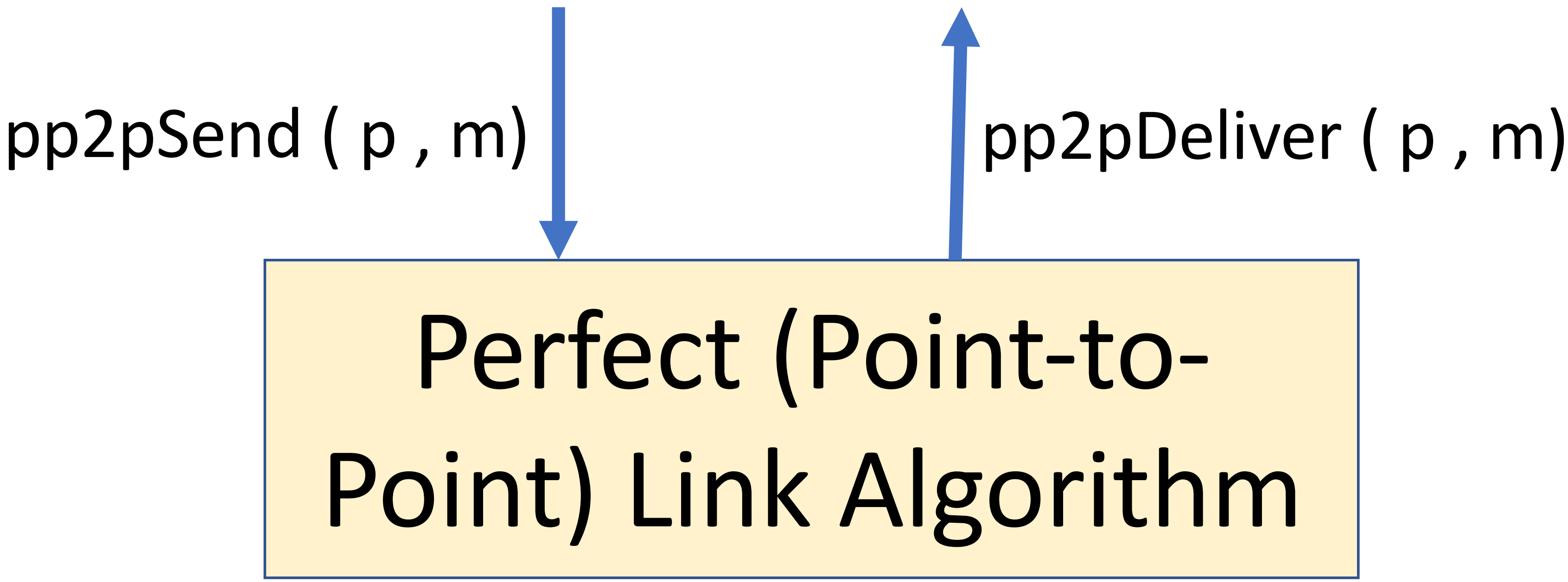
The Anatomy of an Algorithm:



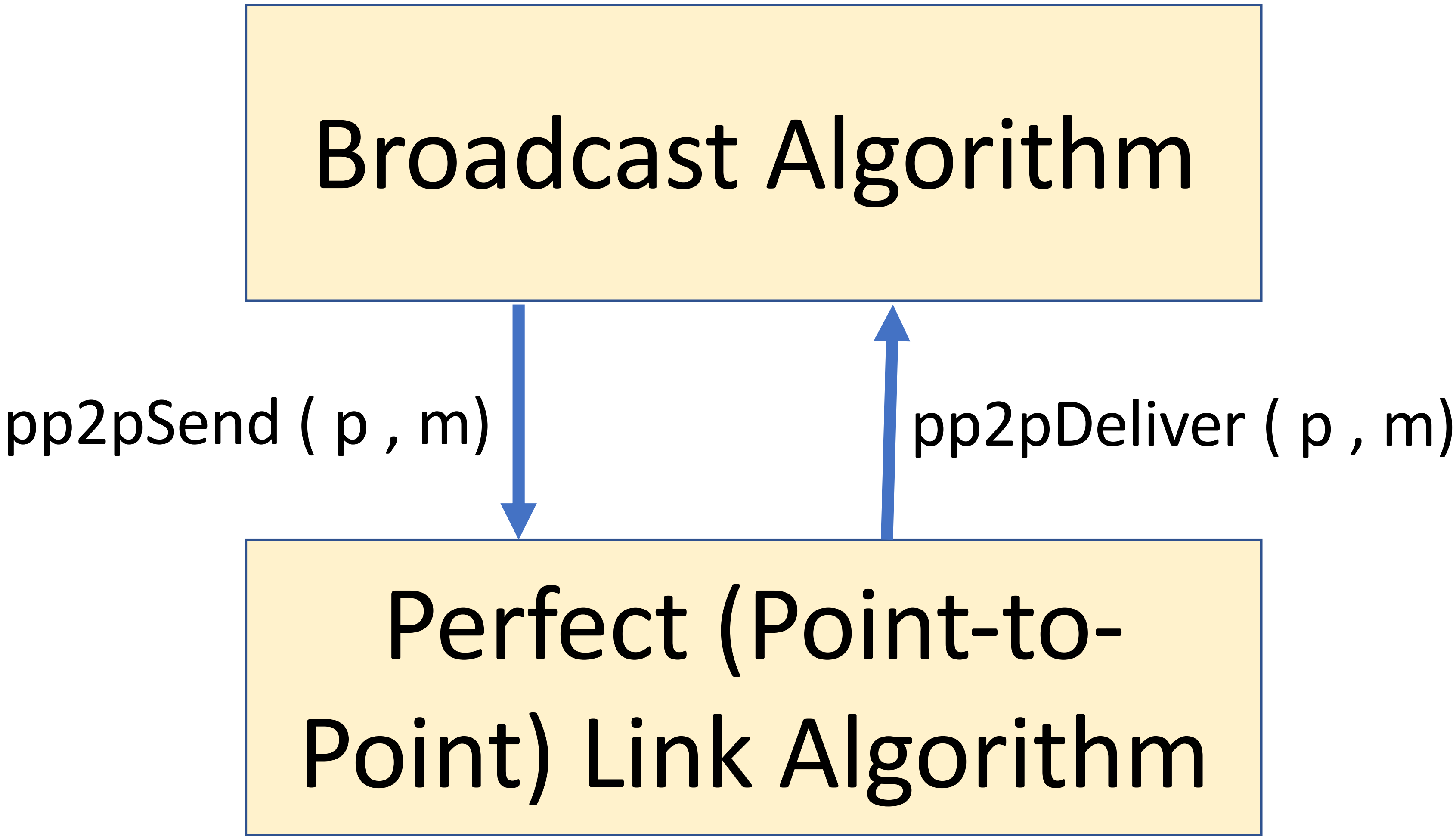
You need an Interface for this to be used (think of APIs)

Let’s use an event driven interface.

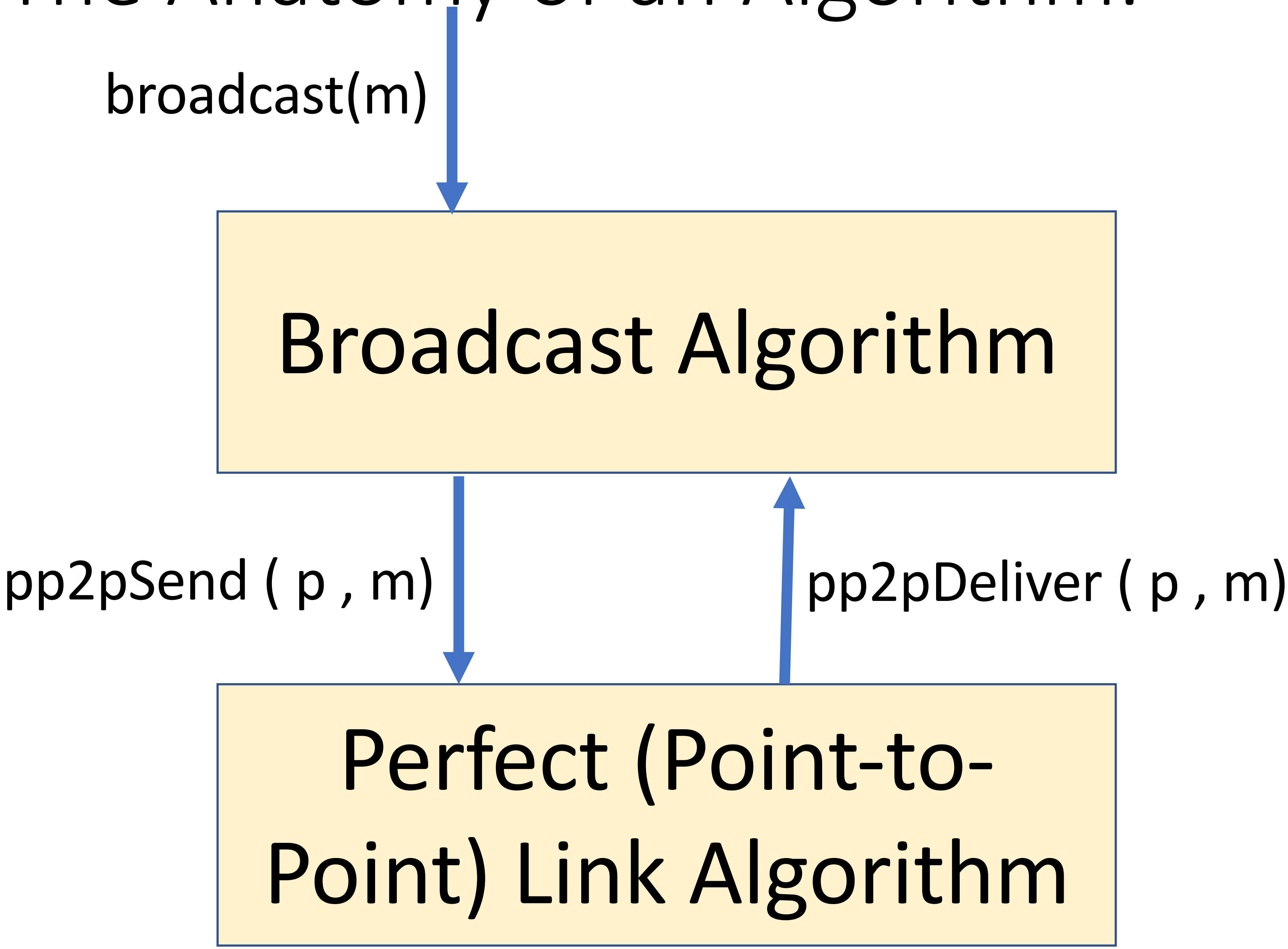
The Anatomy of an Algorithm:



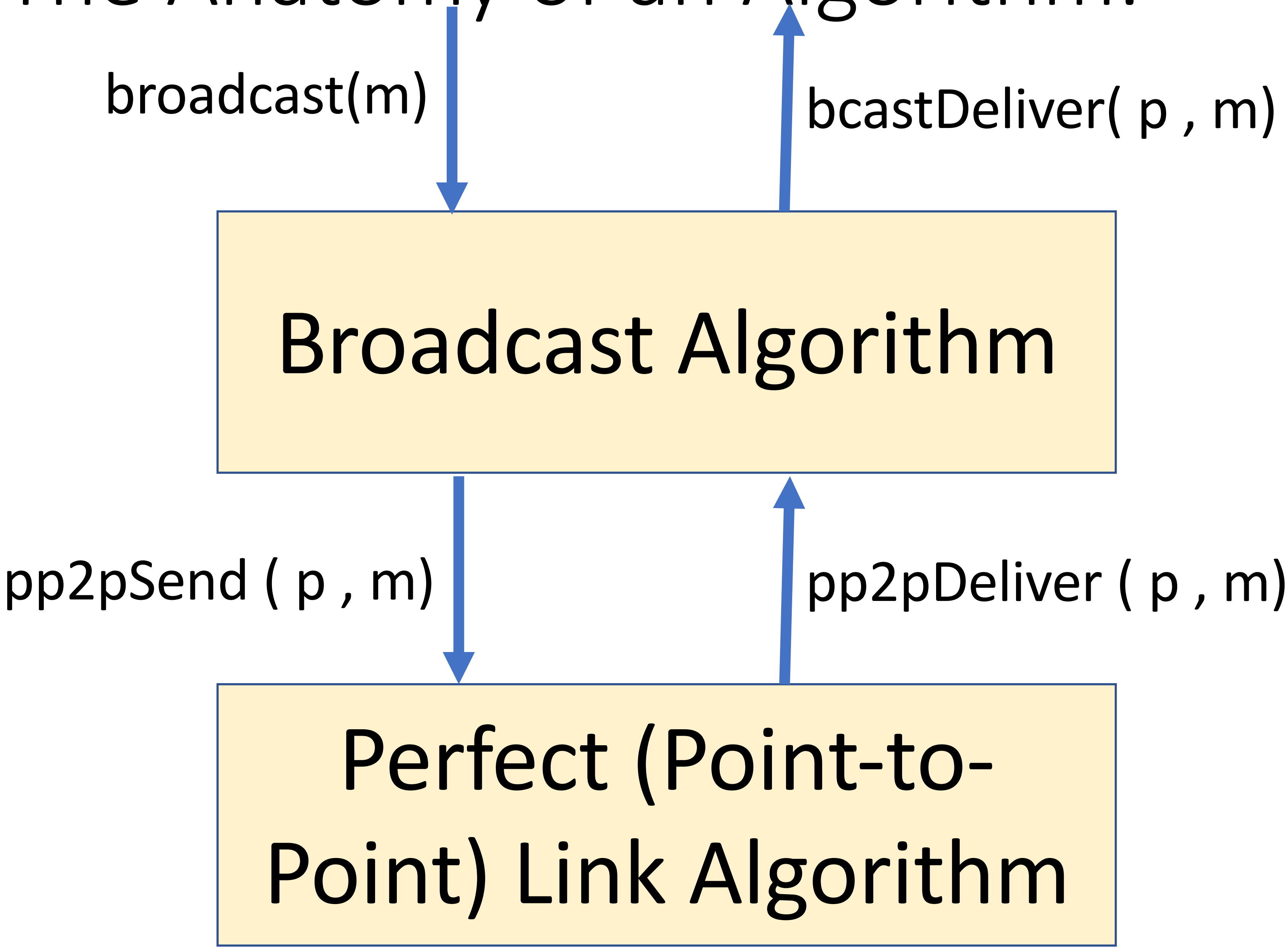
The Anatomy of an Algorithm:



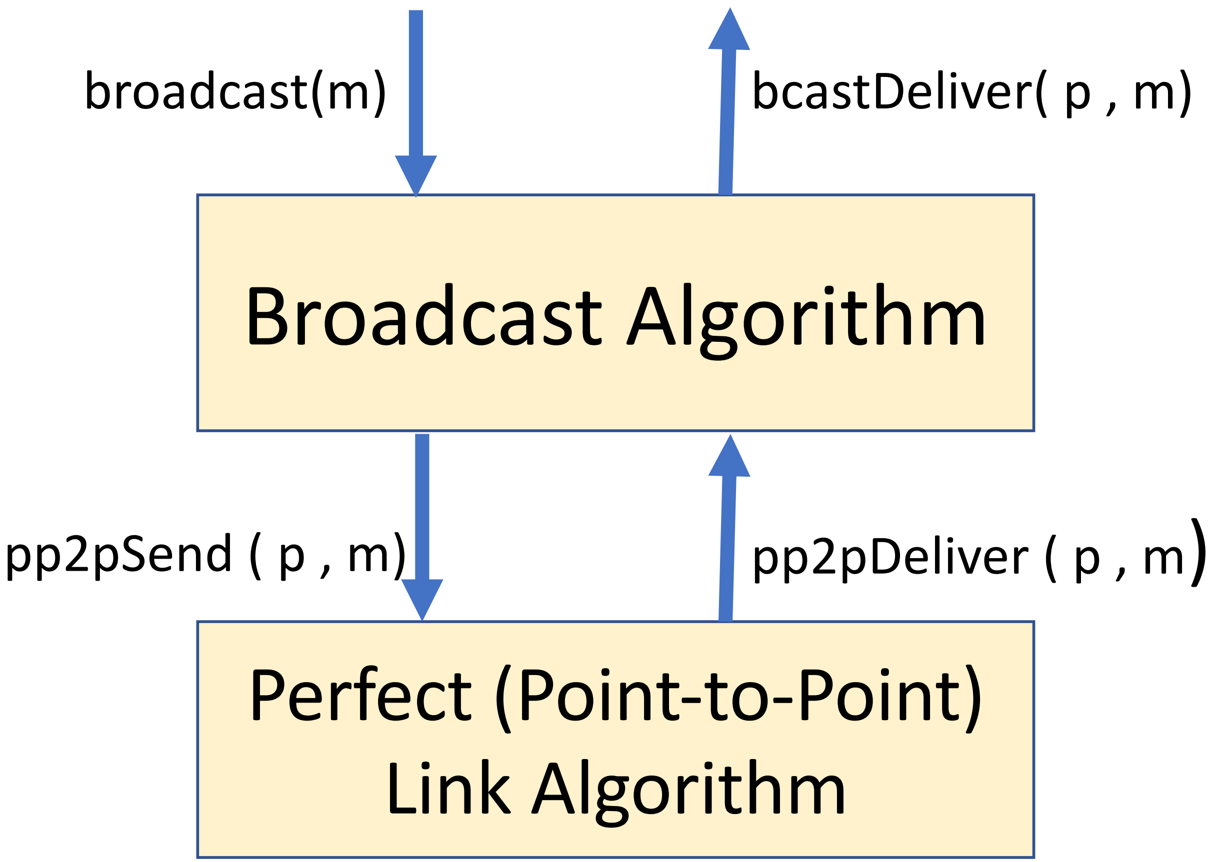
The Anatomy of an Algorithm:



The Anatomy of an Algorithm:



The Anatomy of an Algorithm:



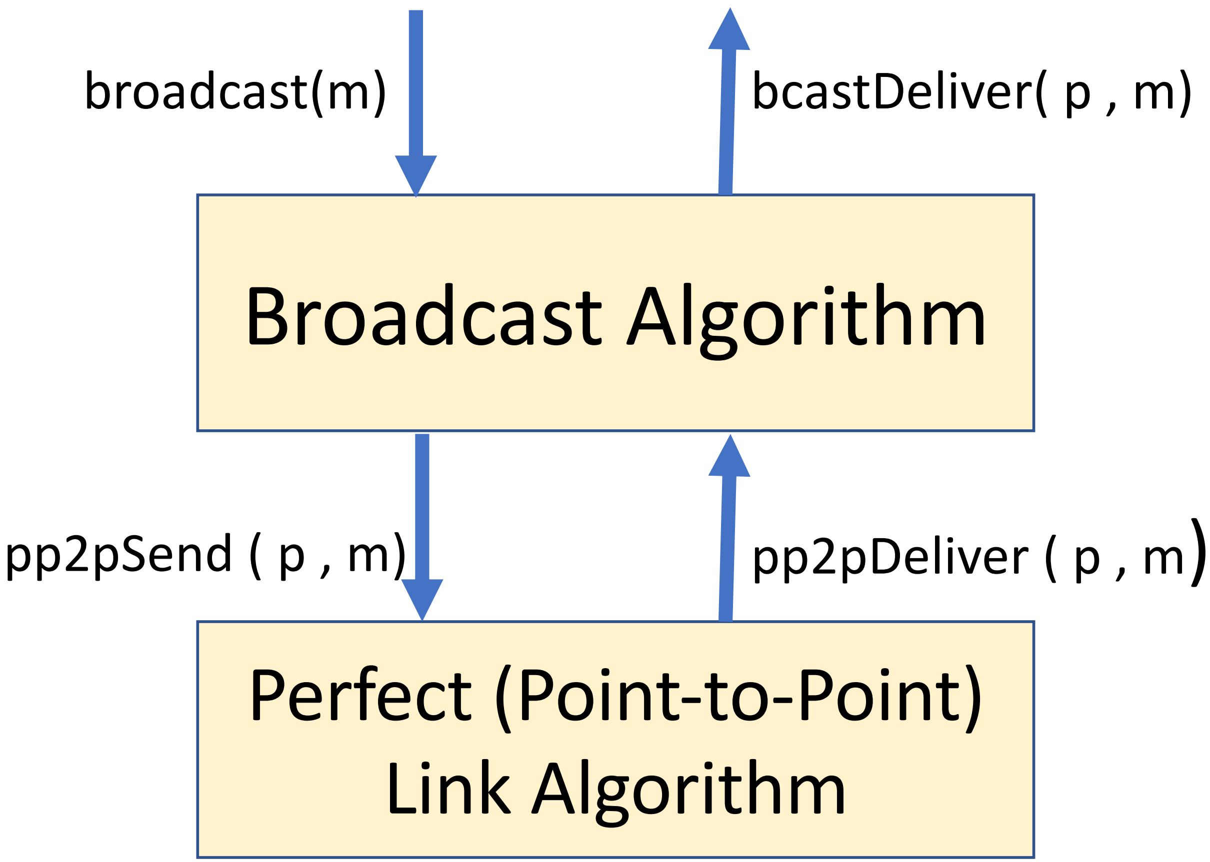
Now you specify what actions are performed when one of this events happen:

Upon broadcast(m) do:

...

...

The Anatomy of an Algorithm:



Now you specify what actions are performed when one of this events happen:

Upon broadcast(m) do:

…

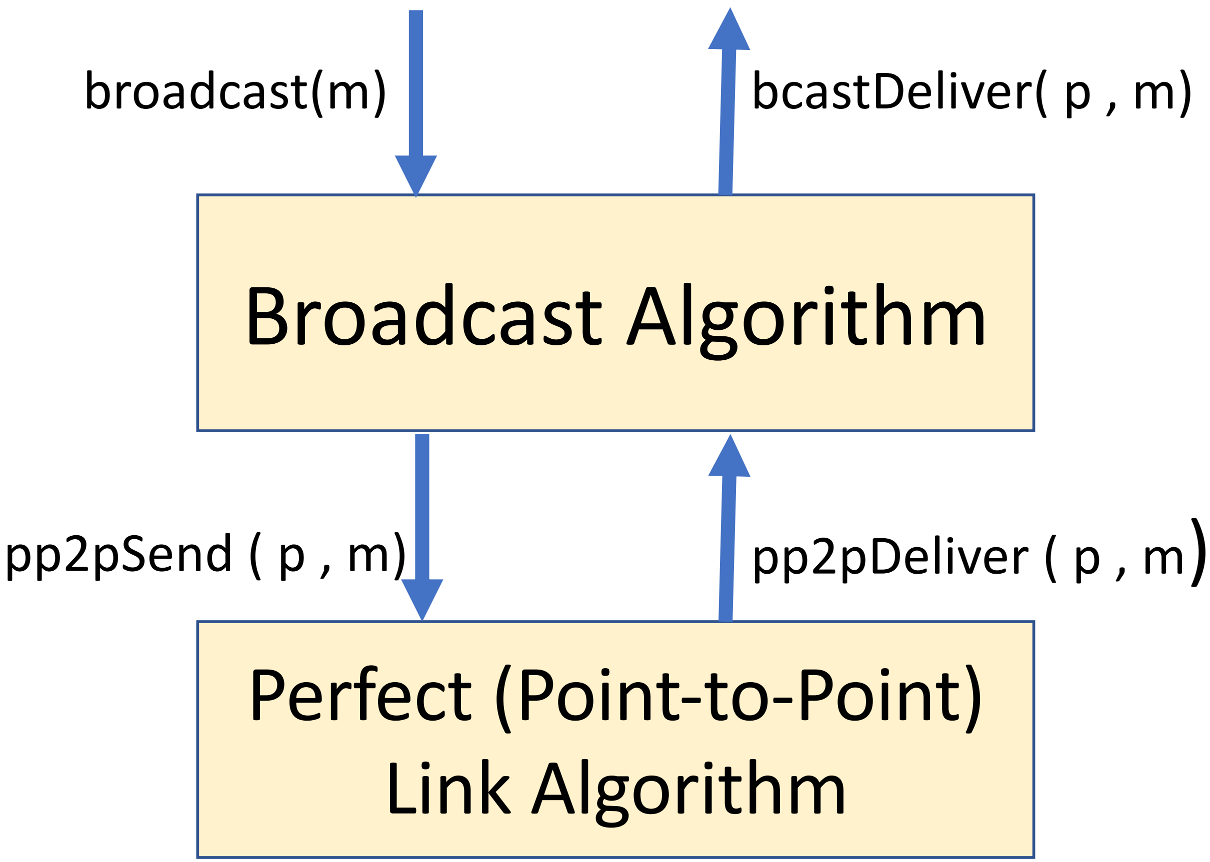
…

You can trigger na event on other protocol:

Trigger pp2psend(p, m)

The Anatomy of an Algorithm:

There is a special event called **Init** (initialization) where you can for instance initialize the local state.



Now you specify what happens when one of this events happen:

Upon broadcast(m) do:

…

You can trigger na event on other protocol:

Trigger pp2psend(p, m);

Back to the Broadcast problem…

- What is the simpliest sollution that you can think of?
- **Think like this:**
  - I am a process, someone asks me to send a message to everyone (me including why not?)
  - I know everyone already so basically what do I do?

Back to the Broadcast problem…

- What is the simpliest sollution that you can think of?

- **Solution:**

Just go ahead and send the message to everyone, one at a time. When you get one of these messages, you just deliver it to the uper layer.

Back to the Broadcast Problem:

- Good... that works... Sort of...
- Actually that is the solution for the Best Effort Broadcast.
- Best Effort Broadcast:
  - BEB1 (Best-Effort validity): For any two correct processes ***i*** and ***j***, every message broadcasted by ***i*** is eventually delivered by ***j.***
  - BEB2: (No Duplication): No message is delivered more than once.
  - BEB3: (No Creation):): If a correct process ***j*** delivers a message *m*, then *m* was **broadcast** to ***j*** by some process ***i.***

Best Effort Broadcast

State: (could be omitted)

Upon Init do: (could be omitted)

Upon bebBroadcast(m) **do**
    forall p ∈ **π**
        trigger pp2pSend(p, m);

Upon pp2pDeliver (p , m) **do**
    trigger bebDeliver(p, m);

Best Effort Broadcast

- Not so great right?
- What happens if a process fails while sending messages to all the other processes?

Best Effort Broadcast

- Not so great right?
- What happens if a process fails while sending messages to all the other processes?
- Answer not everyone gets the message…

Reliable Broadcast Problem

- Lets make this somewhat stronger…
- Reliable Broadcast:
  - RB1 (Validity): If a correct process i broadcasts message m, then i eventually delivers the message.
  - RB2 (No Duplications): No message is delivered more than once.
  - RB3 (No Creation): If a correct process j delivers a message m, then m was broadcast to j by some process i.
  - RB4 (Aggrement): If a message m is delivered by some correct process i, them m is eventually delivered by every correct process j.

How do you think we could solve this problem? Remember that we can use the Best Effort Broadcast Algorithm to solve this one…

- Lets make this somewhat stronger…
- Reliable Broadcast:
  - RB1 (Validity): If a correct process i broadcasts message m, then i eventually delivers the message.
  - RB2 (No Duplications): No message is delivered more than once.
  - RB3 (No Creation): If a correct process j delivers a message m, then m was broadcast to j by some process i.
  - RB4 (Aggrement): If a message m is delivered by some correct process i, them m is eventually delivered by every correct process j.

Home Work 1:

- Write the Pseudo-Code for solving the Reliable Broadcast Problem assuming:
  - A Crash fault model and an asynchronous system.
  - Your solution **must** ensure all properties of the Reliable Broadcast Problem.

  - Remember that if a process crashes he is no longer correct, and hence messages sent by him can never be delivered (as long as no-one delivered those messages).