Failure Detection

In order for a system to appropriately react to failures, failures should be detected in a **timely** manner. A faulty process might get contacted even though it won't be able to respond, increasing latencies and reducing overall system availability.

Failures may occur on **the link level** (messages between processes are lost or delivered slowly), or **on the process level** (the process crashes or is running slowly), and **slowness** may not always be distinguishable from **failure**. This means there's always a **trade-off** between wrongly **suspecting alive processes as dead** (producing false-positives), and **delaying marking an unresponsive process as dead**, giving it the benefit of doubt and expecting it to respond eventually (producing false-negatives).

A **failure detector** is a local subsystem responsible for identifying failed or unreachable processes to exclude them from the algorithm and guarantee liveness while preserving safety.

Liveness and safety are the properties that describe an algorithm's ability to solve a specific problem and the correctness of its output. More formally, **liveness** is a property that guarantees that **a specific intended event must occur**. For example, if one of the processes has failed, a failure detector must detect that failure. **Safety** guarantees that **unintended events will not occur**. For example, if a failure detector has marked a process as dead, this process had to be, in fact, dead.

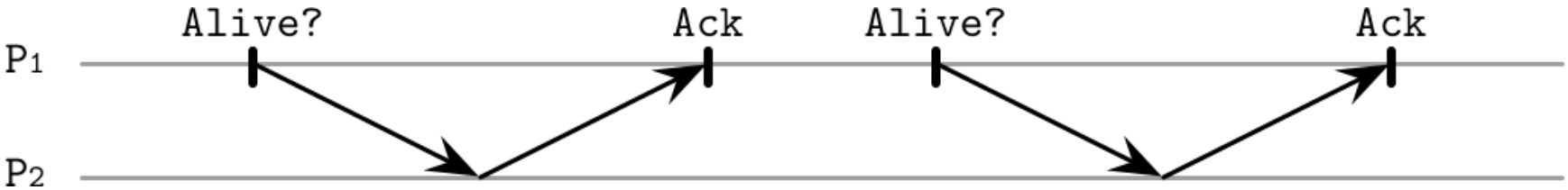
Many distributed systems implement failure detectors by using **heartbeats**. This approach is quite popular because of its simplicity and strong completeness. Algorithms we discuss here assume the absence of Byzantine failures: **processes do not** attempt to intentionally **lie** about their state or states of their neighbors.

Heartbeats and Pings

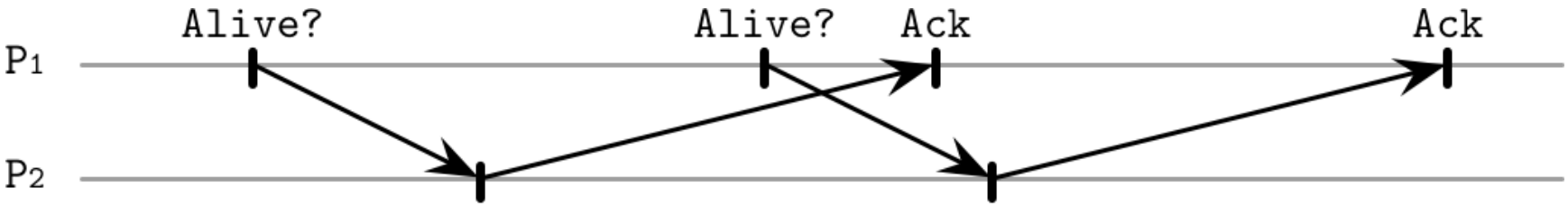
We can query the state of remote processes by triggering one of two periodic processes:

- We can trigger a **ping**, which sends messages to remote processes, checking if they are still alive by **expecting a response** within a specified **time period**.
- We can trigger a **heartbeat** when the process is actively **notifying its peers** that it's still running by sending messages to them.

We'll use **pings** as an example here, but the same problem can be solved using heartbeats, producing similar results.



Pings for failure detection: normal functioning, no message delays



Pings for failure detection: responses are delayed, coming after the next message is sent

Gossip and Failure Detection

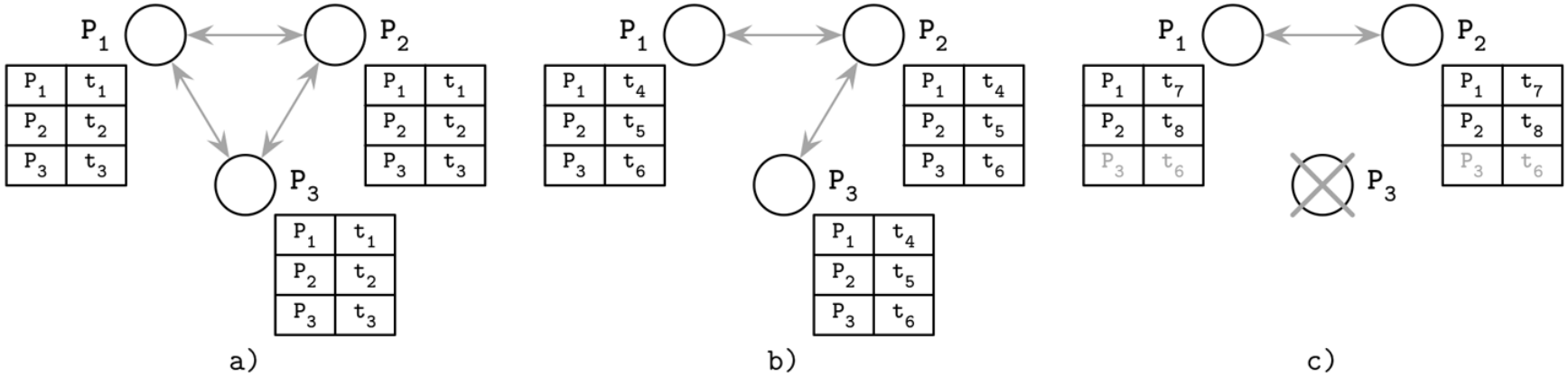
It avoids relying on a single-node view to make a decision.

Each member maintains a list of other members, and timestamps, specifying when the heartbeat was incremented for the last time. Periodically, each member updates its list and distributes it to neighbors. Upon the message receipt, the neighboring node merges the list with its own, updating the list for the other neighbors.

Nodes also periodically check the list. If any node has no updates for long enough, it is considered failed. This timeout period should be chosen carefully to minimize the probability of false-positives.

As the graph below shows

a) All three can communicate and update their timestamps.
b) P3 isn't able to communicate with P1, but its timestamp t6 can still be propagated through P2.
c) P3 crashes. Since it doesn't send updates anymore, it is detected as failed by other processes.



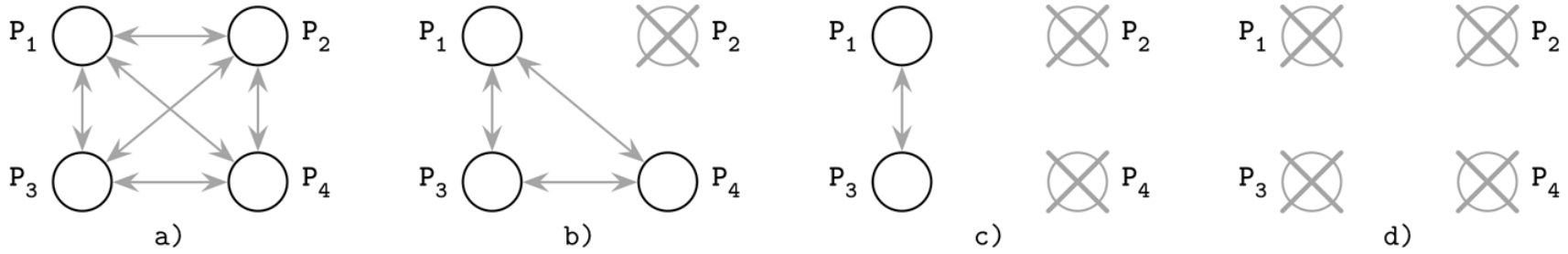
Reversing Failure Detection Problem Statement

To detect process failures, this approach arranges all active processes in **groups**. If one of the groups becomes unavailable, all participants detect the failure. In other words, every time a **single process failure** is detected, it is converted and propagated as a **group failure**. This allows detecting failures in the presence of any pattern of disconnects, partitions, and node failures.

Processes in the group periodically send ping messages to other members, querying whether they're still alive. If one of the members cannot respond to this message because of a crash, network partition, or link failure, the member that has initiated this ping will, in turn, stop responding to ping messages itself.

The graph below shows four communicating processes:

a) Initial state: all processes are alive and can communicate.
b) P2 crashes and stops responding to ping messages.
c) P4 detects the failure of P2 and stops responding to ping messages itself.
d) Eventually, P1 and P3 notice that both P1 and P2 do not respond, and process failure propagates to the entire group.



Here, we use the absence of communication as a means of propagation. An advantage of using this approach is that every member is guaranteed to learn about group failure. One of the downsides is that a link failure separating a single process from other ones can be converted to the group failure as well, but this can be seen as an advantage, depending on the use case.

Applications

- MongoDB

Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible.

- Spark

Executors keep sending metrics for active tasks to the driver every `spark.executor.heartbeatInterval`. Heartbeats let the driver know that the executor is still alive and update it with metrics for in-progress tasks.

*References

Chapter 9. Failure Detection, Database Internals: A Deep Dive into How Distributed Data Systems Work by Alex Petrov
https://spark.apache.org/docs/latest/configuration.html
https://docs.mongodb.com/manual/core/replica-set-elections/#heartbeats