

# Chapter 5

# Elections

Peer-to-Peer systems:

all systems are equal – no requirement for “a leader”

Hierarchical systems:

some special process provide a requirement for Co-ordination

**Examples:**

Replicated services to ensure consistency

Reliable services

Self organizing services

Group co-ordination

All of these could be done in advance if the systems were reliable. But distributed systems need to be dynamic.

If you have thousands of computing resources on many sites – the failure of one cannot prevent the operation of the whole system.

## **Sharing tasks**

Assume we have a problem that can be divided into a number of tasks which do not need to communicate – except at the end of the process.

Any optimisation where you try a set of values, calculate an outcome and record all the outcomes.

Weather forecasting

Travelling salesman

Parameter scan

We could take the number of tasks divide it by the number of processors and submit that number to each processor. Then wait for them all to finish and return the job to a single place to analyse.

“The system which co-ordinates the results is a single point of failure”

“A single slow machine will slowdown the whole calculation”

### **Static sharing**

We could take the number of tasks divide it by the number of processors and submit that number to each processor. Then wait for them all to finish and return the job to a single place to analyse.

“The system which co-ordinates the results is a single point of failure”

“A single slow machine will slowdown the whole calculation”

“A crashed machine will mean missing results”

“A crashed analysis machine means all results are lost”

- 1)“The system which co-ordinates the results is a single point of failure”
- 2)“A single slow machine will slowdown the whole calculation”
- 3)“A crashed machine will mean missing results”
- 4)“A crashed analysis machine means all results are lost”

### **Dynamic sharing**

A co-ordinator reads a task and sends it to an “empty” machine.

The co-ordinator may act as the analyser or delegate the job to another processor.

The co-ordinator only hands out the next task when a machine is empty – solving 2

The co-ordinator checks the status of machines and determines if one has crashed – solving 3

The co-ordinator checks the status of the analysis machine and appoints a new one if it crashes – solving 4

To solve 1 the machines must be able to identify a new co-ordinator.

The analyser is just a different sort of co-ordinator and could be appointed in the same way as the co-ordinator.

How to co-ordinate an “unreliable system” in presence of failures.

Decisions must be made by an arbitrary group as to which process will co-ordinate (some part of ) the system.

Referred to as an *Election*.

Occurs on start-up or when the co-ordinator (also called master) suffers a failure

### **Assumptions**

Any process may call an election

A process may call at most one election at a time.

Multiple processes may call an election simultaneously (*simultaneously is hard to define in a distributed system*).

### **Requirements**

The result does not depend on who initiates the election

*Safety*: one non-failed machine is selected

*Liveness*: Elections terminate

**Constraints:**

1. *Scalability*: the system must work for 1 to  $n$  processes, where  $n$  is arbitrarily large
  2. *Robustness*: the system must work in the presence of failures.
  3. *Efficiency*: not too much communication (or CPU usage) during normal operation or during negotiation
  4. *Responsiveness*: the absence of a master process should be detected rapidly and a new process should be speedily promoted
  5. *Uniqueness*: multi-master situations should be avoided or detected and resolved speedily
  6. *Spurious*: false detection of the circumstance to trigger an election should be avoided
  7. *Independent*: it must run independently in all the processes, without central coordination.
- The implication is that the number and size of the messages required to complete the election does not depend strongly on the number of processes

### **Bully algorithm**

1. Each machine has some sort of *ID*, which is unique and ordered.
2. Each machine knows about all the other machines and their IDs
3. Which machines are active at the start of the election is unknown

The machine with the highest ID is the coordinator.

The co-ordinator fails – what next?

Failure needs to be detected.

Co-ordinator sends a short message to all machines regularly (*heartbeat*). A machine which notices that the message is absent then starts an election.

Interval depends on requirement.

For our example since the master is only required when a machine is empty and needs another task (or a job finishes and needs to be sent to the analyser) – this would be a suitable mechanism.

### **Bully algorithm**

A machine notices that the master is missing. It sends an ***election*** message to all machines with a higher ID.

*Notice this communication must be non-blocking.*

If no reply is received during a time-out period –  $2 * (\text{transit time}) + (\text{processing time})$  it sends a ***coordinator*** message to all processes with a lower ID telling them it is the co-ordinator.

If it receives a reply (it must be from a higher ID), it waits for receipt of a ***coordinator*** message. (it sets a time out on the wait for the coordinator message and if it does not materialise will start a new election).

A process which receives an ***election*** message sends a reply and starts a new election – unless it is already holding one.

A process holds an election on start or re-start.

A process which has noticed the lack of a coordinator – and starts an election, but then discovers it does not have the highest ID, will normally wait for the coordinator message before returning to work

Correctness

*One might argue that if it is the second highest ID, it will know that it must be the co-ordinator and does not need to send an election message*

Don't make a special case and complicate the algorithm.

**Does it work and is it time consuming?**

If the second highest ID, notices a problem it sends a message to the highest ID

It sends an election message to the highest ID and after time-out sends N-2 coordinator messages.

***Takes 1 time-out (round-trip + processing time) and time to send N-2 messages***

The lowest ID notices the fault it sends an election message to all the other machines.

**All** other machines reply and start an election

Those N-2 machines send (N-3), (N-4), (N-5), ... messages so  $\Sigma (N-2)$  **election** messages.

(N-3), (N-4), (N-5), ..... replies.

Total =  $O(N^2)$

and then N-1 co-ordinator messages.

**Best case**

**Worst case**

Elections

**Does it work and is it time consuming?****Worst case**

Second highest machine learns about things after 1 message transit time.

Needs to start an election and wait for the highest ID to time out (2 transit plus processor time)

(Replies to other machines while waiting)

Then sends a co-ordinator message to all other machines.

Other machines hear after 4 message transit times plus a processor time.

Of course not all the machines hear about the new co-ordinator at the same time and all are waiting until they do.

**Best case** other machines do not stop working, they merely receive a new co-ordinator message

**Does it work and is it time consuming?**

Other machines start their election and receive replies and pause while waiting for the coordinator message.

**Worst case***Considerations*

When a coordinator fails normally more than 1 machine will start an election (in response to election requests from a lower ID).

This means we must expect machines to be responding to more than one election request  
So the timeout response time must allow for a machine which is busy answering messages and so the time out must allow for that.

## What happens if two machines decide they are co-ordinators?

Two sets of coordinator messages are sent out.

Other machines have to decide which one to believe.

The one with the highest ID - is the obvious one.

But perhaps the message from the highest ID is actually from a machine which has subsequently crashed.

Tries to communicate with **1**  
**1** times out

Machine starts an election – **2** replies – and starts an election (unless it is already holding one).

But it isn't holding an election – it has sent out coordinator message to everyone.

Need to timestamp communications

How do we timestamp a communication?

A distributed clock –

Chapter Distributed time.

A particular example of a the problems which an unreliable network and unreliable processors can produce.

Most consensus algorithms require that the system is stable for some minimum time in order for it to arrive at a decision.

Proceedings of the International Conference on Autonomic Computing 17-18  
May 2004. R. J. Anthony  
DOI: 10.1109/ICAC.2004.1301356

Introduces two new states.

Normal: Master; Slave

New: Candidate; Idle

**Normal States**

Master:

- Co-ordinate the host services
- Send out heart-beat (beacon messages to inhibit election)

Slave:

- Monitors state of Master
- Performs allocated tasks
- Elects new master when required

**Novel States:**

Candidate:

- Election participation

Slave:

Demotes from slave if too many slaves

Idle:

- Monitors Slave population
- Performs allocated tasks
- Promotes to slave if not enough slaves

**Operation i)**

Slaves monitor master heart-beat and when it disappears only the slaves enter the candidate state in which the election occurs.

A master is chosen, the slaves and master wait in the candidate state for a time for few master heartbeats to check the old master is really gone.

The new master takes over and the system returns to a working state.

Candidate elections take the same length of time independent of the idle pool.

As long as the election algorithm is sound the system will appoint only one master and given the size of the idle pool is approximately constant the accuracy of the algorithm can be tested under all realistic conditions.

**Operation ii)**

Slaves have two extra tasks: they send out slave heartbeats and listen for slave heartbeats.

Since the number of slaves is deliberately kept small these represent only a small extra load.

Idle machines monitor the slave heartbeat.  
Small overhead and the work involved scales with the size of the system.

When an idle machine registers that there are too few slaves it promotes itself to a slave state.

If a slave registers there are too many slaves it drops back into an idle state.

The size of the slave pool remains approximately constant.

**Operation iii)**

The trigger lower limit for pool size can be made different to the upper limit. (Stops flapping).

The trigger conditions (eg time without a slave heartbeat which causes a transition) can be different for different machines.

Thus we don't get every idle (or slave) machine making the transition simultaneously; prevents waves of activity sweeping across the system.

Conditions can be varied to discourage the system from entering the candidate state in the presence of say network instability

Advantages

- 1.Scalability:
- 2.Robustness:
- 3.Efficiency:
- 4.Responsiveness:
- 5.Uniqueness:
- 6.Spurious:
- 7.Independent:

Much better election response.  
Fractionally more load under normal conditions

Normal and fault condition

*In a simpler system in the case of a failure in the system in the master state the election involves co-ordination between  $n-1$  other systems. Elections in such systems involve number of messages which increase with some power of the number of processes participating.*

This compromises conditions 1,3 and 4 of our constraints on slide 7 – (possibly also 2)

Elections are particularly susceptible to instabilities, which may prevent the system choosing a new master.

Limiting the number of participants helps with all these problems.

Small overhead in terms of extra slave heartbeats. Slaves need to listen to slave heartbeats, in addition to their normal job.

Idle processes would in any case be listening to master heartbeats and here they merely listen to slave ones.