CES-27 & CE-288
Distributed Programming

Chapter 5 – Election algorithms

Celso Massaki HIRATA

hirata@ita.br

Instituto Tecnológico de Aeronáutica

Agenda

1. Introduction

2. Chang-Roberts algorithm

3. Ring algorithm

4. The Bully algorithm

1. Introduction

Election

- An **election algorithm** aims to c**hoose a** unique **process** to **play** a particular **role**.
  - **Electing a leader/coordinator process** is an important **task** in a larger context of **distributed processing.**
- **A leader election algorithm** is generally used in conjunction with other distributed algorithms**.**

Election

- A **process calls the election** to proceed its execution.
- The process **initiates** a particular **run** of the **election algorithm**
- An individual **process does not call more** than **one election** at a **time**
  - ✓ But in principle the ***N*** **processes** can call ***N*** **concurrent elections**

Algorithms of Election

- In this **chapter**, we will see three algorithms.
  - ✓ **Chang** and **Robert's** algorithm
  - ✓ **Ring** algorithm
  - ✓ **Bully** algorithm
- The first two employ **the ring topology.**

2. Chang and Robert's algorithm

Chang and Robert's algorithm

- **Two stage ring-based election algorithm** used to find a **process** with the **largest identification**
  - The elected **leader**

- It is a useful method in **decentralized distributed computing**.
  - The **algorithm** works for **any number** of processes
  - The algorithm **does not** require any **process** to know **how many processes** are in the **ring.**

Assumptions

- Each **process** has a **U**nique **Id**entification (UID)
- Unidirectional **ring topology**
  - With a **communication channel** going **from** each **process** to the **clockwise neighbor**.
- We assume that **no failure** occurs
- The system is **asynchronous**

Goal & stages

- **Goal** of the algorithm
  - ✓ **Electing** a **single process** called the **coordinator**
    - The **process** with the **largest identifier**.
- **Stages** of the algorithm
  - ✓ The algorithm is divided in two stages

First stage of the algorithm

1. Initially **every process** in the **ring** is **marked** as ***non-participant in a election***.
2. A **process** that **notices** a **lack** of **leader starts** an **election**.
    1. It creates an **election message** containing its **UID**.
    2. It then **sends** this **message** to its **neighbor**.
3. Every time a process **sends** or **forwards** an **election message**
    - ✓ The process also **marks itself** as a **participant**.

First stage of the algorithm

4. When a **process receives** an **election message** it compares the **UID** of the **message** with its **own UID**.

➢ **Four cases** must be considered (next **slides**):

4.1. If the arrived **UID is larger**

4.2. If the arrived **UID** **is smaller** and the **receiver** is **not** a **participant**

4.3. If the arrived **UID** **is smaller** and the receiver is already a **participant**

4.4. If the **UID** in the incoming **election message** is the **same** as the **UID** of the **process**

First stage of the algorithm

4. When a **process receives** an **election message** it compares the **UID** of the **message** with its **own UID**.

4.1. If the arrived **UID is larger**

- The process forwards the **election message** in a **clockwise** direction.

4.2. If the arrived **UID is smaller** and the **receiver** is **not** a **participant**

- The process **replaces** the **UID** in the **message** with its **own UID**
- The process **sends** the **updated election message** in a **clockwise** direction.

First stage of the algorithm

4. When a **process receives** an **election message** it compares the **UID** of the **message** with its **own UID (cont.)**

4.3. If the **UID is smaller**, and the **process** is already a **participant**

- The **process** has already sent out an **election message** with a **UID** at least as **large** as its **own UID**
- The process **discards** the **election message**.

4.4. If the **UID** of the **incoming election message** is the **same** as the **UID** of the process

- That **process starts acting** as the **leader**.

Second stage of the algorithm

1. The **leader process**
   - ✓ Marks itself as **non-participant**
   - ✓ Sends an **elected message** to its neighbor announcing its election and **UID**.
2. When a **process** receives the **elected message**
   - ✓ It marks itself as **non-participant**
   - ✓ It records the **elected UID**
   - ✓ It forwards the **elected message** unchanged.

Second stage of the algorithm

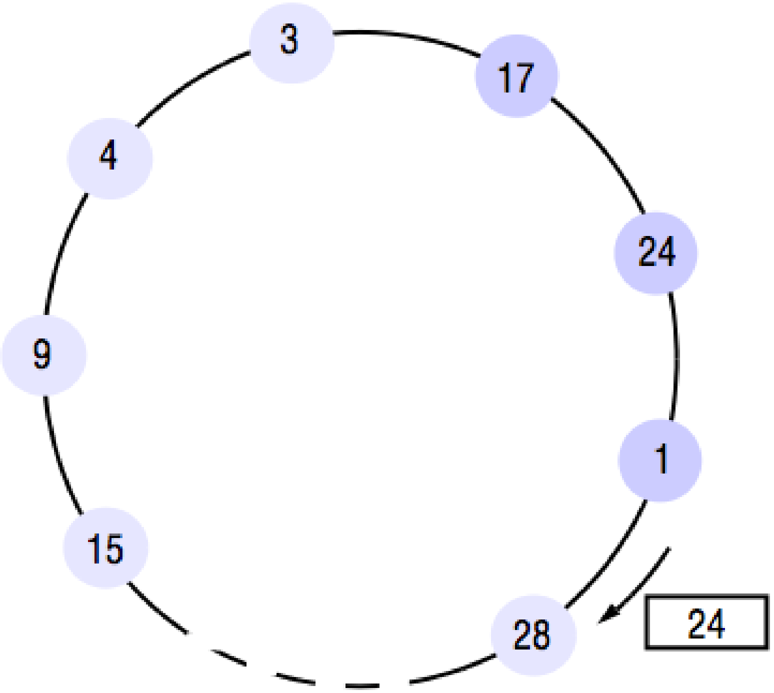
3. When the **elected message** reaches the **elected leader:**
   - ✓ The **leader discards** that **message**
   - ✓ The **election** is **over**.

(Why the second stage?)

- ➢ Assuming **there is no failure,** this **algorithm** will **finish**.

Example of ring-based election in progress

- The **election** was started by **process 17**
  - ✓ The highest **UID** found so far is **24**
  - ✓ **Participants processes** are shown in **dark** fill color (17, 24, and 1)



Safety & liveness properties

- The **algorithm** respects “**safety”**:
  - A **process** will receive an **elected message** with its own **UID** only if his **UID** is **greater than** other **UIDs** and only when all processes agree on the same **UID**.

Safety & liveness properties

- The **algorithm** also respects “**liveness**”
  - “**participant**" and "**not participant**" **states** are used
  - When **multiple processes start an election** at roughly the same time, **only a single winner** will be **announced**

Performance analysis and fault tolerance

- When there is a **single process** starting the **election**, the **algorithm** requires ***3.n-1*** **sequentially messages**, in the **worst case** (***n*** is the number of processes)
- The **worst case** is when the **process** starting the **election** is the **immediate following** to the one with **greatest UID**.

- The **algorithm** is **not fault tolerant**.

Ring algorithm

Ring algorithm

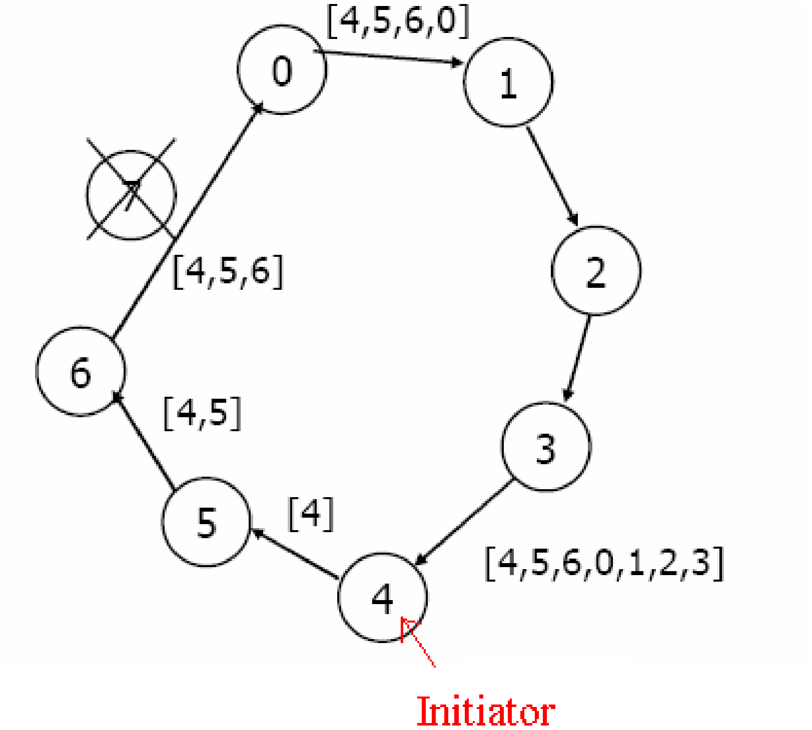
- When a process notices that coordinator is not working:
  - Builds an ELECTION message (containing its own process number)
  - Sends the message to its successor. If successor is **down, sender skips over it** and goes to the next member along the ring, or the one after that, until a running process is located. We assume that there is a way to detect that the successor is down.
  - At each step, sender adds its own process number to the list in the message.

Ring algorithm

- When the message gets back to the process that started it all:
  - ✓ Process recognizes the message that contains its own process number
  - ✓ Changes message type to COORDINATOR
- Circulates message once again to inform everyone else: who the new coordinator is (list member with highest number); **who the members of the new ring are**.
- When message has circulated once, it is removed.

Ring algorithm

- **Performance**
  - ✓ ***2.n*** messages in the worst case, ***n*** is the number of active processes.



3. Bully algorithm

Coordinator election

- The **Bully algorithm elects a new coordinator** in the possibility of **coordinator failure**.
  - It **dynamically elects** a **coordinator** by **process ID** number.
    - The **process** with the **highest process ID number** is **selected** as the **coordinator**.

Assumptions

- Communication is of form send-receive
- **Timeout** is used for identifying **process failure**
- **Message delivery** between processes is **reliable**

Bully vs. Chang and Robert's algorithm

- Bully **assumes** that communication is of form "**send - initiates timer – receive**"

- Bully uses **timeout** to **detect** process **failure/crash**

- Each **process** knows which **processes** have the **higher identifier numbers** and **communicates** with **them.**

Message Types

- **Election Message**:
  - ✓ Sent to **announce** the **election**
- **Answer Message**:
  - ✓ Respond to the **election message**
- **Coordinator message**:
  - ✓ Sent to **announce** the identity of the **new elected process**
    - **Coordinator process**

Failure detection

- A process P_i may determine that the current **coordinator** is **down**
  - ✓ Based on **message timeout**
  - ✓ Based on **failure** of the ***Pi*** to initiate a handshake

- To do this, a process P_i performs the **four actions**

Bully Algorithm

1. **P_i broadcasts** an **election message** (inquiry) to all other **processes** with **higher** process **IDs**.
2. If $\boldsymbol{P_i}$ hears from **no process** with a higher process **ID** than it, it **wins** the **election** and **broadcasts victory**.

Bully Algorithm

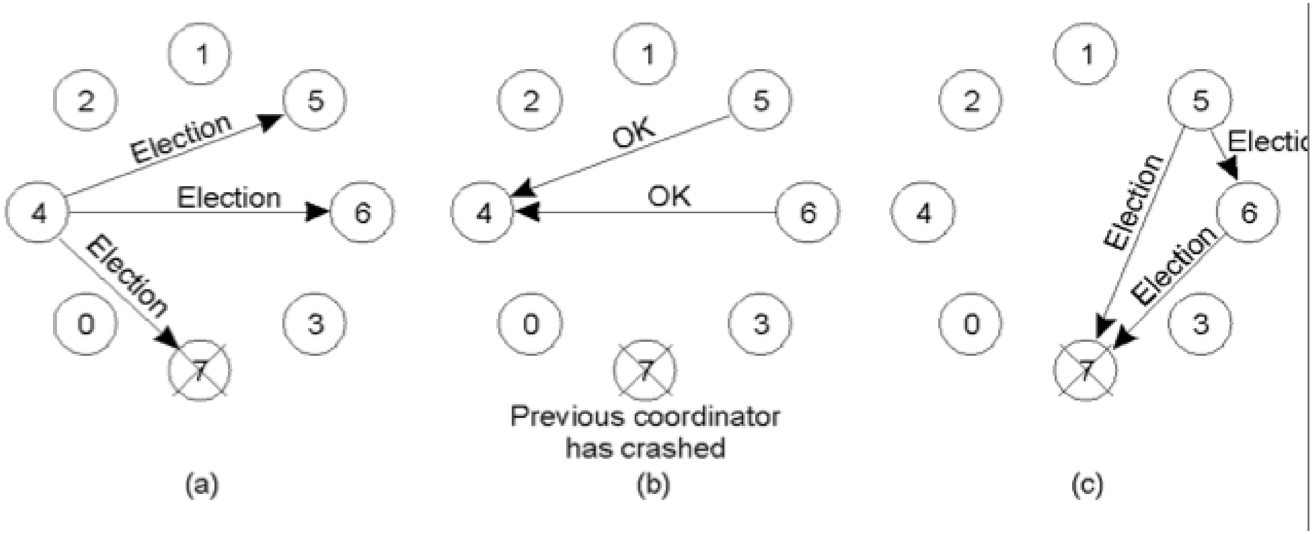
3. If P_i hears from a **process** with a **higher ID**, P_i waits a certain amount of time for that process to **broadcast** itself as the **leader**.
   - ✓ If it **does not receive** this **message** in time, it **re-broadcasts** the **election message**.
4. If P_i gets an **election message** from another process with a **lower ID** it sends an **"I am alive" message back** and **starts** new **election**.

Bully Algorithm

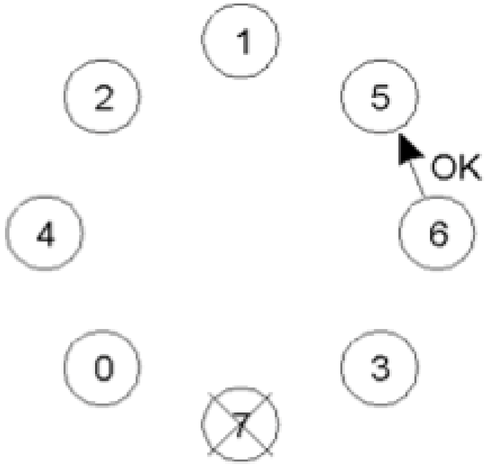
- Note that if P_i receives a **victory message** from a **process** with a **lower ID** number, it immediately initiates a **new election**.
- This is how the **algorithm** gets its name
  - ✓ A **process** with a **higher ID number** will **bully** a **lower ID process** out of the **coordinator position** as soon as it comes **online**.

Example

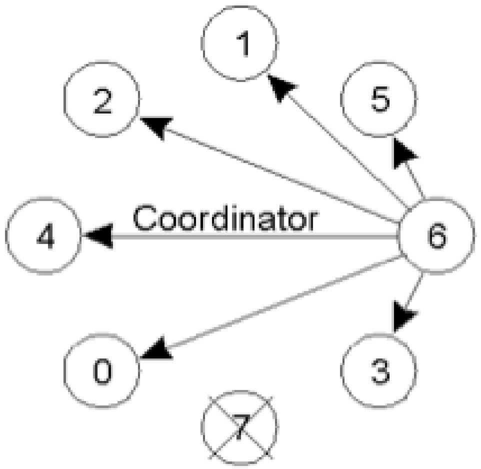
- **Process 4** holds an **election**
  - ✓ Processes **5** and **6** respond, telling **4** to stop
  - ✓ Now **5** and **6** each holds an election



Example



(d)



(e)

Performance analysis

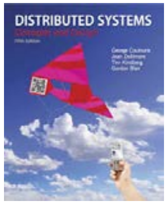
- **Best case**
  - The process with the **second-highest identifier** notices the **coordinator's failure**
  - Then it can immediately elect itself and send ***N*–2 coordinator messages**

- **Worst case**
  - $O(\boldsymbol{n}^2)$ messages
  - When the process with the **lowest identifier** first detects the **coordinator's failure**

References

- Garg, Vijay K. **Concurrent and distributed computing in Java**; **chapter 13**: Leader election. John Wiley & Sons, 2005.

- Coulouris, George F., Jean Dollimore, and Tim Kindberg. **Distributed systems: concepts and design**; **chapter 15**: coordination and agreement. Pearson education, 5th edition, 2012.



References

- Chang, E. and R. Roberts. "**An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processors**." *Comm. ACM* 22.5 (1979).

Operating Systems
R. Stockton Gaines Editor

An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes

Ernest Chang
University of Toronto

Rosemary Roberts
University of Waterloo

Introduction

Given a random circular arrangement of uniquely numbered processes where no a priori knowledge of the number of processes is known, and no central controller is assumed, we would like a method of designating by consensus a single unique process. The algorithm we propose works equally well for finding either the highest numbered or the lowest numbered process. Let us, without loss of generality, consider highest finding.

A situation in which this algorithm is important has been presented by LeLann [1]. In his example, a circle of controllers in which the control token is lost causes every controller to time out, and an election to find a new emitter for the control token is performed. LeLann's algorithm requires every controller to send a message bearing its number. Each controller thus collects, through the messages seen, the numbers of the other controllers in the circle. Every controller sorts its list, and the controller whose own number is the highest on its list is elected.

LeLann's algorithm, in a circle with n controllers, requires total messages passed proportional to n^2, written $O(n^2)$, where a message pass is a SEND of a message from a controller. This is clearly so, since each of the n controllers sends a message which is passed to all other nodes. Our algorithm requires, on the average, $O(n \log n)$ message passes.

References: Bully algorithm

- Garcia-Molina, Hector. "**Elections in a distributed computing system**." *Computers, IEEE Transactions on* 100.1 (1982): 48-59.

Elections in a Distributed Computing System

HECTOR GARCIA-MOLINA, MEMBER, IEEE

***Abstract*—After a failure occurs in a distributed computing system, it is often necessary to reorganize the active nodes so that they can continue to perform a useful task. The first step in such a reorganization or reconfiguration is to elect a coordinator node to manage the operation. This paper discusses such elections and reorganizations. Two types of reasonable failure environments are studied. For each environment assertions which define the meaning of an election are presented. An election algorithm which satisfies the assertions is presented for each environment.**

***Index Terms*—Crash recovery, distributed computing systems, elections, failures, mutual exclusion, reorganization.**

out to *reorganize* the system. During the reorganization period, the status of the system components can be evaluated, any pending work can either be finished or discarded, and new algorithms (and possibly a new task) that are tailored to the current situation can be selected. The reorganization of the system is managed by a *single* node called the *coordinator*. (Having more than one node attempting to reorganize will lead to serious confusion.) So as a first step in any reorganization, the operating or active nodes must *elect* a coordinator. It is precisely these elections we wish to study in this paper.