Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Broadcast Algorithms
Logical Clocks
Distributed Semaphores

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Broadcast Algorithms
Logical Clocks
Distributed Semaphores

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Networks in distributed environments

- Systems in distributed environment can be at the same time
  - Server
  - Client
- In distributed systems no one server has all the information needed for servicing clients
- They exchange data and work together to service clients

Broadcast primitive is needed!

- **Transmits** a message from one process/processor to **all others**
  - Very useful programming technique in distributed systems

```
broadcast ch(m)
```

- Places the message `m` on `n` channels `ch`
  - `n`: number of receiving process
  - `ch`: channel (better ch_n) for each receiving process

- Reference: Andrews p.383

Broadcast primitive

`broadcast ch(m)` is like:

- Places one copy of the message *m* on each channel ch_i
  - Including his own channel
- Or in other words: Executing *n* `send` statements

- Client's receiving is as usual: `vars = receive(ch`$_i$`)`

Challenges and usage of broadcast

- Broadcast messages from two processes A and B might be received by other processes in **different orders!**
  - **Synchronization is problematic**

Whiteboard

Example:

- The broadcast primitives could be used to create **distributed semaphores**, for example
  - Total ordering of communication events is the basis
    →Logical clocks

Challenges and usage of broadcast

- Broadcast messages from two processes A and B might be received by other processes in **different orders!**
  - **Synchronization is problematic**

Whiteboard

Example:

- The broadcast primitives could be used to create **distributed semaphores**, for example
  - Total ordering of communication events is the basis
    →Logical clocks

Broadcast – Exercise

Try to write a broadcast primitive in Python

→ Please refer to Ex23, Question 2

Key take away messages

- Broadcast primitive sends the message to all
  `broadcast ch(m)`
- Order of different broadcast messages can vary
- Used e.g. for distributed semaphores

Next to come:
Broadcast Algorithms
Logical Clocks
Distributed Semaphores

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Broadcast Algorithms
Logical Clocks
Distributed Semaphores

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Logical clocks & Event ordering

In a distributed program:

- Processes execute local actions and communication actions
  - Local actions: e.g., reading/writing local variables
    - No direct effect on other processes
  - Communication actions: e.g., sending/receiving messages
    - These affect the execution of other processes, since messages are used to synchronize distant processes

Events

- Communication actions are significant ***events*** in distributed programs

- **`send`** and **`receive`** statements can be seen as ***events***

Event ordering (1/2)

- Two process A and B execute local action → no chance to know the relative order of the action's execution
- If messages are exchanged → the order is known
  - Sending is done before receiving

- A sends to B, B waits for msg from A, B then sends to C
  - We have a **total order** :

    A→B, B→C ➔ A→B→C

Event ordering (2/2)

- There is a total order between events that work together
- But total order between entire collection of events in a distributed program is not guaranteed
  - E.g., no ordering among different sets of processes in the communications between the processes

Thus, we need a single central clock

Single central clock

- Each message is combined with a **timestamp** (tick)
- Send: append to the message a timestamp from the *single central clock*
- Receive: get the message and the timestamp
- Ordering: messages by timestamps
- Typically, NTP Service (Network Time Protocol)
  - Service enabling clients across Internet to be synchronized accurately to UTC (atomic clock)
  - NTP employs statistical technique for the filtering of timing data between the quality of timing data from different servers

Single central clock, the problem

Single central clock doesn't exist in distributed systems

Why?

- Different processes on the same processor can be perfectly synchronized, they use the same clock source
- But distributed processes in a network have **never perfectly** synchronized clocks!

→ Other solution needed: **Logical clock**

Logical clock

- G. Andrews and Lamport Logical Clock [1978]:
- Integer counter
- Incremented when **event** occurs
  - Remember that events are *send* and *receive*
- Each process saves the **logical clock** (lc)
- Each message contains as an addition a **timestamp**

Logical clock – Incrementation

Logical Clock update rules (Andrews version)

- send/broadcast some_channel(msg, lc)
  lc+=1

- (msg, ts) = receive some_channel
  lc=max(lc, ts+1)
  lc+=1

Logical clock – Example

Andrews version

Process A	lc	Process B	lc
Init of the system	0	*Init of the system*	0
`send (msg, lc);` `lc++`	0 1		

		`receive (msg, ts) //ts=?`
		`lc=Max(ts+1, lc);`
		`lc++`	

		`Send (msg, lc);` `lc++;`	

`receive (msg, ts) //ts=?`
`lc=Max(ts+1, lc);`
`lc++`			

Logical clock – Example

Andrews version

Process A	lc	Process B	lc
Init of the system	0	*Init of the system*	0
send (msg, lc); lc++	0 1		0

	1	receive (msg, ts) //ts=0
		lc=Max(ts+1, lc);
		lc++
			0
			1
			2

	1	Send (msg, lc); lc++;	2 3

receive (msg, ts) //ts=2
lc=Max(ts+1, lc);
lc++
	1
	3
	4		

Logical clock – Incrementation

Logical Clock update rules (Lamport Algo)

- lc++
- send/broadcast some_channel(msg, lc)
- (msg, ts) = receive some_channel
  lc=max(lc, ts)+ 1

Logical clock – Example

Lamport Logical Clock

Process A	lc	Process B	lc
Init of the system	0	*Init of the system*	0
`lc++` `send (msg, lc);`	1 1		
		`receive (msg, ts) //ts=?` `lc=Max(ts, lc)+1;`	
		`Send (msg, lc);`	
`receive (msg, ts) //ts=?` `lc=Max(ts+1, lc)+ 1;`			

Logical clock – Example

Lamport Logical Clock

Process A	lc	Process B	lc
Init of the system	0	*Init of the system*	0
lc++ send (msg, lc);	1 1		

		receive (msg, ts) //ts=1
		lc=Max(ts, lc)+1;
			0
			2

		Send (msg, lc);	2

receive (msg, ts) //ts=2
lc=Max(ts, lc)+ 1;
	1
	3		

Logical clock – Exercise

Exercise 23, Question 3 → Ex23 Link

- Play processor and determine what are possible end values of the logical clocks of the 3 processes?

Key take away messages

- Events – **`send`** and **`receive`**
- Event ordering not known between different processes
- Logical clock
  - `send/broadcast some_channel(msg, lc); lc++;`
  - `receive some_channel(msg, ts); lc=max(lc, ts+1); lc++;`

Next to come:
Broadcast Algorithms
Logical Clocks
Distributed Semaphores

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Broadcast Algorithms
Logical Clocks
Distributed Semaphores

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Distributed semaphores

- Semaphores use normally shared variables
- Semaphores can also be implemented with **message passing** (→ seen in lectures before, *active monitor*)

- The processes that **“share**” a semaphore need **to cooperate** so that they **maintain the semaphore state** even though the program state is **distributed!**

DistributedSem – Technique (1/2)

- Processes broadcast (also to itself) messages for `P()` and `V()` with
  - Sender's identity
  - Tag (P, V, others)
  - Timestamp (logical clock)
- Processes examine the messages they receive to determine when to proceed

DistributedSem – Technique (2/2)

- Each process needs a message queue ***mq***
- Each process needs a logical clock ***lc***

- The process stores the message in the queue
  - The queue is kept **sorted** (increasing message's timestamp)
  - Sender's IDs are used to break ties if needed
    - ID is then used as priority level (lower ID → higher priority)

DistributedSem – Acknowledgment of messages

- Received messages (P or V) are acknowledged by everybody
  - Broadcast message
  - Has a timestamp
  - However, ACKs are **not** entered in the message queue (mq)
  - ACKs are not acknowledged again → otherwise infinite loop! ☹
- **ACK helps to determine when a regular message in *mq* has become fully acknowledged and thus every process processed it**

DistSem – Example

- *Visit the sample code in the lecture documents*

- The *User* process initiates **V()** and **P()** operations by broadcasting messages on the *semop* channels
- The *Helper* process implements the **V()** and **P()** operations

- Per participant:
  - one *Helper-process*
  - one *User-process*

Key take away messages

- Distributed semaphores use
  - broadcast message passing
  - logical clock
- Used to synchronize distributed processes
- Logical clock
  - `send/broadcast some_channel(msg, lc); lc++;`
  - `receive some_channel(msg, ts); lc=max(lc, ts+1); lc++;`