CM3 Distributed Computing

Part 1: Intro & Message Passing Systems

Tom Friedetzky

tom.friedetzky@dur.ac.uk
Office E250

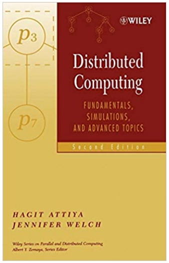
Outline

Distributed Computing

Tom Friedetzky
Office: E250
E-mail: tom.friedetzky@durham.ac.uk

DUO
Lecture slides, formative exercises. . . AND! MORE!!!

Textbook
Attiya and Welch: Distributed Computing
ISBN 0471453242



Introduction

A *distributed system* is a collection of individual computing devices that can communicate with each other.

Processors must coordinate their actions in order to perform one large task.

Examples

- VLSI chip
- shared memory multiprocessor
- local-area cluster of workstations
- Internet

Advantages of distributed systems

- share resources
- share data
- communicate
- increase performance speed
- increase tolerance of failures

So distributed systems are desirable. . . but constructing a functioning system is very difficult!

Pragmatic Difficulties

- heterogeneous hardware
- heterogeneous software
- lack of universal standards

Fundamental Difficulties

- asynchron(icit)y
- limited knowledge
- failures
- attacks

We focus on some fundamental difficulties

How to overcome difficulties

Construct a *theoretical framework*:

- identify and abstract fundamental problems
- state problems precisely
- design algorithms or show that it is impossible to solve problems
- prove correctness of algorithms
- analyse complexity of algorithms

Advantages of this approach:

- good abstraction describes reasonably many situations
- discover *inherent* limitations

Complications of this approach for distributed systems:

- no single, universally accepted model of computation
- major differences between distributed systems depending on
  - how processors communicate
  - what kind of timing information is available
  - what kind of failures are to be tolerated.
- communication costs besides computation time and space

Because of these complications: there is a large interest in *"negative results"*, e.g.,

Prove that a particular problem

- cannot be solved in a particular kind of distributed system, or
- can only be solved with a certain amount of some resource.

This is useful information for a designer!

On the positive side:

It is often possible to solve a problem after slightly changing some assumptions

Because there are many models in the area of distributed computing:

- this submodule is organised around fundamental problems
- we consider three main models

Fundamental problems

In order to show the underlying concepts we consider a number of fundamental problems.

For each problem we indicate how the choice of model influences its solvability or complexity.

Application areas that have provided fundamental problems in distributed computing:

- operating systems
- distributed databases
- software fault-tolerance
- the internet
- multiprocessor architectures.

Main Models

Each model we consider is a combination of

- a basic communication model
- a basic timing model

Communication models

We consider two basic *communication* models:

- *message passing*: processors communicate by sending messages over communication channels
- *shared memory*: processors communicate via a common memory area that contains a set of shared variables

Timing models

There are two basic *timing* models:

- *asynchronous:* no fixed upper bound on process execution speeds and message delivery delays.
- *synchronous:* fixed upper bound on process execution speeds and message delivery delays.

Example of asynchronous system. Internet: e-mail messages usually take seconds to arrive but can take days.

Combined models

We have

- **the synchronous message passing model**
- **the asynchronous message passing model**

- the synchronous shared memory model
  (e.g., PRAM; a little in TCS3)
- **the asynchronous shared memory model**

Syllabus (give or take. . .)

	Book	
Lectures	Chapter	Topic
1,2	2	algorithms in message passing systems
3–5	3	the leader election problem
6,7	4	mutual exclusion in shared memory
8	5	fault tolerance
9,10	6	causality and time

Summative Assessment
Coursework & end-of-year exam

Formative Assessment
Bits & pieces throughout term

Learning outcomes

Be able to understand and explain

- the basic models of distributed systems and fundamental problems in distributed computing.
- how distributed systems work
- why they sometimes do not work
- the fundamental difficulties in distributed computing
- how theory of design and analysis of algorithms can be used to solve computational problems

Outline

Message Passing

In a *message passing* system processors communicate by sending messages over *(communication) channels*.

Processors are denoted by $p_1, p_2, \ldots, p_n$.

A channel between p_i and p_j is denoted by (p_i, p_j).

Assumption: Channels are bi-directional: if (p_i, p_j) is a channel:

- p_i is able to send messages to p_j and
- p_j is able to send messages to p_i

A graph G with vertices $p_1, \ldots, p_n$ and edges (p_i, p_j) whenever there is a channel between p_i and p_j is called a *(network) topology graph*.

Example of a problem

Let $T = (V_T, E_T)$ be a topology tree on n nodes with root p_r.

Say p_r has some information that needs to be sent as a message m to all other processors in T.

How can this be done?

This problem is called the *single message broadcast problem*.

How do we solve it?

Example of an algorithm

Spanning tree broadcast algorithm

```
Code for $p_r$
upon receiving no message:
    send $m$ to all children
    terminate
```

```
Code for $p_i, \ i \neq r$
upon receiving $m$ from parent:
    send $m$ to all children
    terminate
```

This is a correct algorithm (obvious).

How good is it?

Complexity measures

There are two complexity measures. Both are *worst-case*.

To define these measures we assume:

- each processor is able to terminate.
- once terminated a processor remains terminated.

Then an algorithm has *terminated* when every processor has terminated.

Complexity measures:

1. *message complexity*: the maximum number of messages sent in any execution of the algorithm.
2. *time complexity*: the maximum *time* taken by any execution of the algorithm.

How do we define “time”?

Asynchronous systems

In *asynchronous* message passing systems there are usually upper bounds on message delays.

They can be very large but may, in practice, not occur that often.

We want an algorithm to be independent of them.

Time is expressed in terms of maximum message delays.

For convenience we set the maximum message delay equal to one unit of time.

Synchronous systems

In *synchronous* message passing system algorithms are executed in *rounds*.

In each round:

- each processor can send a message to each neighbour;
  (some models: much more restrictive)
- messages sent in a round are delivered in the same round;
- each processor can perform computations based on the messages just received.

In synchronous systems the time complexity is the maximum number of rounds of any execution of the algorithm.

Let $T = (V_T, E_T)$ be a topology tree on n nodes with root p_r.

The *depth* d of T is the maximum distance between p_r and any other processor.

Theorem

In both the asynchronous and synchronous model the message complexity of the Spanning tree broadcast algorithm is $n - 1$.

Proof

In both models exactly one message is passed over each channel. Then, in both models, the total number of messages sent during the algorithm is $|E_T| = n - 1$.

Theorem

In both the asynchronous and synchronous model the time complexity of the Spanning tree broadcast algorithm is d.

Proof

Synchronous model:
It is clear that after exactly d rounds message m has reached all processors.

Asynchronous model:
Let P be a path from p_r (root) to a maximum-distance leaf in T.
Suppose there is a maximum message delay on each edge in P.
Then the total time is $|E_P| = d$.
It is clear that this is the worst-case scenario.

Note on synchronous systems

In practice systems (that are not closely coupled) usually do not perform in rounds.

So synchronous systems are **far** from realistic.

Why do we still study them?

- synchronous systems are much *easier* to design and understand (less uncertainty).
- if it is difficult or impossible to solve a problem on a synchronous system, then it is even harder to solve this problem on an asynchronous system
- in practice systems are not completely asynchronous either; an algorithm designed for the synchronous model can be simulated to work in a more realistic model

Constructing a Spanning Tree

We will discuss the *Flooding algorithm* (**2.3**).

This algorithm constructs a spanning tree of a given connected topology graph.

Learning outcomes:

- to gain a better understanding of the
  - models,
  - correctness arguments,
  - complexity measures

  for distributed algorithms.

The Single Message Broadcast problem

Let $G = (V_G, E_G)$ be a connected topology graph on n nodes.

Let $p_r \in V_G$ have some information M that needs to be sent as a message $\langle M \rangle$ to all other processors in G.

(So $\langle M \rangle$ denotes a message with content M.)

We call p_r the *specified root* of G.

If G is a tree, we use the *Spanning tree broadcast algorithm*.

(see earlier)

What if G is not a tree?

What if p_r needs to broadcast some messages $\langle M_2 \rangle, \langle M_3 \rangle, \langle M_4 \rangle, \ldots$ as well?

A solution

We can do as follows:

1. Construct a spanning tree T of G with root p_r while passing $\langle M \rangle$.
2. Use the *Spanning tree broadcast algorithm* with input T to broadcast the messages $\langle M_2 \rangle, \langle M_3 \rangle, \ldots$

Recall that a subgraph T of a connected graph G is called a spanning tree if T is a tree with $V_T = V_G$.

There are several algorithms that construct a spanning tree T of a connected topology graph G.

We consider the *Flooding algorithm*. This algorithm can be used in both the synchronous and asynchronous model.

See **2.4** for another algorithm.

Flooding algorithm (code for each p_i)

```
Initially $parent = NIL$, $Children = \emptyset$, $Other = \emptyset$
  **if** receiving no message **then**
    **if** $p_i = p_r$ and $parent = NIL$ **then**
      send $\langle M \rangle$ to all neighbours
      $parent := p_i$
  **if** receiving $\langle M \rangle$ from neighbour $p_j$ **then**
    **if** $parent = NIL$ **then**
      $parent := p_j$
      send $\langle parent \rangle$ to $p_j$
      send $\langle M \rangle$ to all neighbours except $p_j$
    **else**
      send $\langle already \rangle$ to $p_j$
  **if** receiving $\langle parent \rangle$ from neighbour $p_j$ **then**
    add $p_j$ to $Children$
    **if** $Children \cup Other = N(p_i) \setminus \{parent\}$ **then**
      terminate
  **if** receiving $\langle already \rangle$ from neighbour $p_j$ **then**
    add $p_j$ to $Other$
    **if** $Children \cup Other = N(p_i) \setminus \{parent\}$ **then**
      terminate
```

Remark 1. $N(p_i)$ denotes set of all neighbours of p_i.

Remark 2. $\langle parent \rangle \langle already \rangle$ messages must be send. Otherwise processors do not know which of their neighbours in G are their children in the constructed spanning tree.

Correctness

Let G be a connected topology graph with specified root p_r.

Theorem

The Flooding algorithm constructs a spanning tree of G with root p_r.

Proof

From the code observe that for every p_i

- *parent* is only changed once.
- $|Children|$ does not decrease.
- $|Other|$ does not decrease.

Hence the algorithm terminates.
Also note that p_i is parent of p_j if and only if p_j is child of p_i.

Proof cont'd

We can define subgraph H of G with:

- V_H : all p_i with $parent \neq NIL$.
- E_H : all (p_i, p_j) such that p_i is parent of p_j.

1. Is $V_H = V_G$?
To obtain a contradiction suppose $V_H \neq V_G$.
Since G is connected, by assumption there exist an edge (p_i, p_j) such that $p_i \in V_H$ and $p_j \notin V_H$.
Since p_i in V_H, p_i has $parent \neq NIL$.
By code: p_i has sent $\langle M \rangle$ to all its neighbours, so also to p_j.
Then, again by the code, p_j has $parent \neq NIL$.
CONTRADICTION!
Hence $V_H = V_G$.

Proof cont'd

2. Does H contain a cycle?
In order to obtain a contradiction suppose

$$p_{i_1} p_{i_2} p_{i_3} \dots p_{i_k} p_{i_1}$$

is a cycle.
By code: A child of p_j receives $\langle M \rangle$ for the first time *after* p_j does.
By definition of H: Each processor is the parent of the next processor in the cycle.
Reasoning above would mean that p_{i_1} receives $\langle M \rangle$ for the first time before p_{i_1} does.
CONTRADICTION!
Hence H does not contain cycle.

From **1.** and **2.** we conclude that H is a spanning tree of G. □

Complexities

Theorem

In both the asynchronous and synchronous model the message complexity of Flooding algorithm is $O(|E_G|)$

Proof

Every message of type

- $\langle M \rangle$
- $\langle parent \rangle$
- $\langle already \rangle$

is sent *at most* twice on each edge.
Hence, the total number of messages that are sent is at most

$$6|E_G| = O(|E_G|).$$

$\text{dist}(p_i, p_j)$ denotes the *distance* between p_i and p_j in G.

$D = \max\{dist(p_i, p_j) \mid p_i, p_j \in V_G\}$ denotes the *diameter* of G.

Theorem

In both the asynchronous and synchronous model the time complexity of the Flooding algorithm is $O(D)$.

Proof

Exercise!

As we saw, the *Flooding algorithm* has the same time and message complexities in both synchronous model and asynchronous model.

What is the advantage if we are allowed to use the synchronous model?

What kind of spanning tree is desirable?

Advantage of synchronous model

The depth d of a spanning tree T is minimal if T is a breadth-first search tree.

Hence, having a breadth-first search tree is desirable!

Theorem

In the synchronous model, the Flooding algorithm constructs a breadth-first search spanning tree of G with root p_r.

This may *not* hold for the asynchronous model.