ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Principles of Distributed Computing

Roger Wattenhofer
wattenhofer@tik.ee.ethz.ch

Spring 2011

Introduction

What is Distributed Computing?

In the last few decades, we have experienced an unprecedented growth in the area of distributed systems and networks. Distributed computing now encompasses many of the activities occurring in today's computer and communications world. Indeed, distributed computing appears in quite diverse application areas: Typical "old school" examples are parallel computers, or the Internet. More recent application examples of distributed systems include peer-to-peer systems, sensor networks, and multi-core architectures.

These applications have in common that many processors or entities (often called nodes) are active in the system at any moment. The nodes have certain degrees of freedom: they may have their own hardware, their own code, and sometimes their own independent task. Nevertheless, the nodes may share common resources and information, and, in order to solve a problem that concerns several—or maybe even all—nodes, coordination is necessary.

Despite these commonalities, a peer-to-peer system, for example, is quite different from a multi-core architecture. Due to such differences, many different models and parameters are studied in the area of distributed computing. In some systems the nodes operate synchronously, in other systems they operate asynchronously. There are simple homogeneous systems, and heterogeneous systems where different types of nodes, potentially with different capabilities, objectives etc., need to interact. There are different communication techniques: nodes may communicate by exchanging messages, or by means of shared memory. Sometimes the communication infrastructure is tailor-made for an application, sometimes one has to work with any given infrastructure. The nodes in a system sometimes work together to solve a global task, occasionally the nodes are autonomous agents that have their own agenda and compete for common resources. Sometimes the nodes can be assumed to work correctly, at times they may exhibit failures. In contrast to a single-node system, distributed systems may still function correctly despite failures as other nodes can take over the work of the failed nodes. There are different kinds of failures that can be considered: nodes may just crash, or they might exhibit an arbitrary, erroneous behavior, maybe even to a degree where it cannot be distinguished from malicious (also known as Byzantine) behavior. It is also possible that the nodes follow the rules indeed, however they tweak the parameters to get the most out of the system; in other words, the nodes act selfishly.

Apparently, there are many models (and even more combinations of models) that can be studied. We will not discuss them in greater detail now, but simply define them when we use them. Towards the end of the course a general picture should emerge. Hopefully!

This course introduces the basic principles of distributed computing, highlighting common themes and techniques. In particular, we study some of the fundamental issues underlying the design of distributed systems:

- Communication: Communication does not come for free; often communication cost dominates the cost of local processing or storage. Sometimes we even assume that everything but communication is free.
- Coordination: How can you coordinate a distributed system so that it performs some task efficiently?
- Fault-tolerance: As mentioned above, one major advantage of a distributed system is that even in the presence of failures the system as a whole may survive.
- Locality: Networks keep growing. Luckily, global information is not always needed to solve a task, often it is sufficient if nodes talk to their neighbors. In this course, we will address the fundamental question in distributed computing whether a local solution is possible for a wide range of problems.
- Parallelism: How fast can you solve a task if you increase your computational power, e.g., by increasing the number of nodes that can share the workload? How much parallelism is possible for a given problem?
- Symmetry breaking: Sometimes some nodes need to be selected to orchestrate the computation (and the communication). This is typically achieved by a technique called *symmetry breaking*.
- Synchronization: How can you implement a synchronous algorithm in an asynchronous system?
- Uncertainty: If we need to agree on a single term that fittingly describes this course, it is probably “uncertainty”. As the whole system is distributed, the nodes cannot know what other nodes are doing at this exact moment, and the nodes are required to solve the tasks at hand despite the lack of global knowledge.

Finally, there are also a few areas that we will not cover in this course, mostly because these topics have become so important that they deserve and have their own courses. Examples for such topics are distributed programming, software engineering, as well as security and cryptography.

In summary, in this class we explore essential algorithmic ideas and lower bound techniques, basically the “pearls” of distributed computing and network algorithms. We will cover a fresh topic every week.

Have fun!

Chapter 1

Vertex Coloring

1.1 Introduction

Vertex coloring is an infamous graph theory problem. It is also a useful toy example to see the style of this course already in the first lecture. Vertex coloring does have quite a few practical applications, for example in the area of wireless networks where coloring is the foundation of so-called TDMA MAC protocols. Generally speaking, vertex coloring is used as a means to break symmetries, one of the main themes in distributed computing. In this chapter we will not really talk about vertex coloring applications but treat the problem abstractly. At the end of the class you probably learned the fastest (but not constant!) algorithm ever! Let us start with some simple definitions and observations.

Problem 1.1 (Vertex Coloring)**.** *Given an undirected graph $G = (V, E)$, assign a color c_u to each vertex $u \in V$ such that the following holds: $e = (v, w) \in E \Rightarrow c_v \neq c_w$.*

Remarks:

- Throughout this course, we use the terms *vertex* and *node* interchangeably.
- The application often asks us to use few colors! In a TDMA MAC protocol, for example, less colors immediately imply higher throughput. However, in distributed computing we are often happy with a solution which is suboptimal. There is a tradeoff between the optimality of a solution (efficacy), and the work/time needed to compute the solution (efficiency).

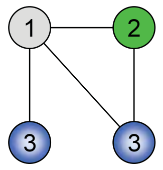

Figure 1.1: 3-colorable graph with a valid coloring.

Assumption 1.2 (Node Identifiers)**.** *Each node has a unique identifier, e.g., its IP address. We usually assume that each identifier consists of only* $\log n$ *bits if the system has* n *nodes.*

Remarks:

- Sometimes we might even assume that the nodes exactly have identifiers $1, \dots, n$.
- It is easy to see that node identifiers (as defined in Assumption 1.2) solve the coloring problem 1.1, but not very well (essentially requiring n colors). How many colors are needed at least is a well-studied problem.

Definition 1.3 (Chromatic Number)**.** *Given an undirected Graph* $G = (V, E)$*, the* chromatic number $\chi(G)$ *is the minimum number of colors to solve Problem 1.1.*

To get a better understanding of the vertex coloring problem, let us first look at a simple non-distributed ("centralized") vertex coloring algorithm:

Algorithm 1 Greedy Sequential

```
1: while $\exists$ uncolored vertex $v$ do
2:     color $v$ with the minimal color (number) that does not conflict with the
       already colored neighbors
3: end while
```

Definition 1.4 (Degree)**.** *The number of neighbors of a vertex* v*, denoted by* $\delta(v)$*, is called the* degree *of* v*. The maximum degree vertex in a graph* G *defines the graph degree* $\Delta(G) = \Delta$*.*

Theorem 1.5 (Analysis of Algorithm 1)**.** *The algorithm is correct and terminates in* n *"steps". The algorithm uses* $\Delta + 1$ *colors.*

Proof: Correctness and termination are straightforward. Since each node has at most Δ neighbors, there is always at least one color free in the range $\{1, \dots, \Delta + 1\}$.

Remarks:

- In Definition 1.7 we will see what is meant by "step".
- For many graphs coloring can be done with much less than $\Delta + 1$ colors.
- This algorithm is not distributed at all; only one processor is active at a time. Still, maybe we can use the simple idea of Algorithm 1 to define a distributed coloring subroutine that may come in handy later.

Now we are ready to study distributed algorithms for this problem. The following procedure can be executed by every vertex v in a distributed coloring algorithm. The goal of this subroutine is to improve a given initial coloring.

Procedure 2 First Free

```
Require: Node Coloring {e.g., node IDs as defined in Assumption 1.2}
  Give $v$ the smallest admissible color {i.e., the smallest node color not used by
  any neighbor}
```

Remarks:

- With this subroutine we have to make sure that two adjacent vertices are not colored at the same time. Otherwise, the neighbors may at the same time conclude that some small color c is still available in their neighborhood, and then at the same time decide to choose this color c.

Definition 1.6 (Synchronous Distributed Algorithm)**.** *In a synchronous algorithm, nodes operate in synchronous rounds. In each round, each processor executes the following steps:*

1. *Do some local computation (of reasonable complexity).*
2. *Send messages to neighbors in graph (of reasonable size).*
3. *Receive messages (that were sent by neighbors in step 2 of the same round).*

Remarks:

- Any other step ordering is fine.

Algorithm 3 Reduce

```
1: Assume that initially all nodes have ID's (Assumption 1.2)
2: Each node $v$ executes the following code
3: node $v$ sends its ID to all neighbors
4: node $v$ receives IDs of neighbors
5: while node $v$ has an uncolored neighbor with higher ID do
6:     node $v$ sends "undecided" to all neighbors
7:     node $v$ receives new decisions from neighbors
8: end while
9: node $v$ chooses a free color using subroutine First Free (Procedure 2)
10: node $v$ informs all its neighbors about its choice
```

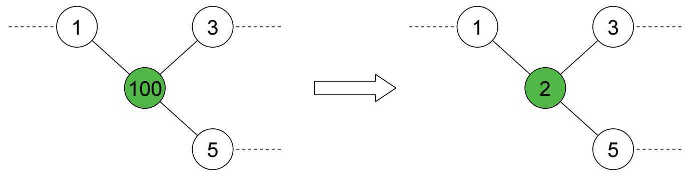

Figure 1.2: Vertex 100 receives the lowest possible color.

Definition 1.7 (Time Complexity)**.** *For synchronous algorithms (as defined in 1.6) the* time complexity *is the number of rounds until the algorithm terminates.*

Remarks:

- The algorithm terminates when the last processor has decided to terminate.
- To guarantee correctness the procedure requires a legal input (i.e., pairwise different node IDs).

Theorem 1.8 (Analysis of Algorithm 3)**.** *Algorithm 3 is correct and has time complexity n. The algorithm uses $\Delta + 1$ colors.*

Remarks:

- Quite trivial, but also quite slow.
- However, it seems difficult to come up with a fast algorithm.
- Maybe it's better to first study a simple special case, a tree, and then go from there.

1.2 Coloring Trees

Lemma 1.9. $\chi(Tree) \leq 2$

Constructive Proof: If the distance of a node to the root is odd (even), color it 1 (0). An odd node has only even neighbors and vice versa. If we assume that each node knows its parent (root has no parent) and children in a tree, this constructive proof gives a very simple algorithm:

Algorithm 4 Slow Tree Coloring

```
1: Color the root 0, root sends 0 to its children
2: **Each node** $v$ concurrently executes the following code:
3: **if** node $v$ receives a message $x$ (from parent) **then**
4:     node $v$ chooses color $c_v = 1 - x$
5:     node $v$ sends $c_v$ to its children (all neighbors except parent)
6: **end if**
```

Remarks:

- With the proof of Lemma 1.9, Algorithm 4 is correct.
- How can we determine a root in a tree if it is not already given? We will figure that out later.
- The time complexity of the algorithm is the height of the tree.
- If the root was chosen unfortunately, and the tree has a degenerated topology, the time complexity may be up to n, the number of nodes.
- Also, this algorithm does not need to be synchronous . . . !

Definition 1.10 (Asynchronous Distributed Algorithm)**.** *In the asynchronous model, algorithms are event driven ("upon receiving message . . . , do . . . "). Processors cannot access a global clock. A message sent from one processor to another will arrive in finite but unbounded time.*

Remarks:

- The asynchronous model and the synchronous model (Definition 1.6) are the cornerstone models in distributed computing. As they do not necessarily reflect reality there are several models in between synchronous and asynchronous. However, from a theoretical point of view the synchronous and the asynchronous model are the most interesting ones (because every other model is in between these extremes).
- Note that in the asynchronous model, messages that take a longer path may arrive earlier.

Definition 1.11 (Time Complexity)**.** *For asynchronous algorithms (as defined in 1.6) the* time complexity *is the number of time units from the start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of at most one time unit.*

Remarks:

- You cannot use the maximum delay in the algorithm design. In other words, the algorithm has to be correct even if there is no such delay upper bound.

Definition 1.12 (Message Complexity)**.** *The* message complexity *of a synchronous or asynchronous algorithm is determined by the number of messages exchanged (again every legal input, every execution scenario).*

Theorem 1.13 (Analysis of Algorithm 4)**.** *Algorithm 4 is correct. If each node knows its parent and its children, the (asynchronous) time complexity is the tree height which is bounded by the diameter of the tree; the message complexity is $n - 1$ in a tree with n nodes.*

Remarks:

- In this case the asynchronous time complexity is the same as the synchronous time complexity.
- Nice trees, e.g. balanced binary trees, have logarithmic height, that is we have a logarithmic time complexity.
- This algorithm is not very exciting. Can we do better than logarithmic?!?

The following algorithm terminates in $\log^* n$ time. Log-Star?! That's the number of logarithms (to the base 2) you need to take to get down to at least 2, starting with n:

Definition 1.14 (Log-Star)**.**
$\forall x \leq 2 : \ \log^* x := 1 \qquad \forall x > 2 : \ \log^* x := 1 + \log^*(\log x)$

Remarks:

- Log-star is an amazingly slowly growing function. Log-star of all the atoms in the observable universe (estimated to be 10^{80}) is 5! There are functions which grow even more slowly, such as the inverse Ackermann function, however, the inverse Ackermann function of all the atoms is 4. So log-star increases indeed very slowly!

Here is the idea of the algorithm: We start with color labels that have $\log n$ bits. In each synchronous round we compute a new label with exponentially smaller size than the previous label, still guaranteeing to have a valid vertex coloring! But how are we going to do that?

Algorithm 5 "6-Color"

```
1: Assume that initially the vertices are legally colored. Using Assumption 1.2
   each label only has $\log n$ bits
2: The root assigns itself the label 0.
3: **Each** other **node** $v$ executes the following code (synchronously in parallel)
4: send $c_v$ to all children
5: **repeat**
6:    receive $c_p$ from parent
7:    interpret $c_v$ and $c_p$ as little-endian bit-strings: $c(k), \dots, c(1), c(0)$
8:    let $i$ be the smallest index where $c_v$ and $c_p$ differ
9:    the new label is $i$ (as bitstring) followed by the bit $c_v(i)$ itself
10:   send $c_v$ to all children
11: **until** $c_w \in \{0, \dots, 5\}$ for all nodes $w$
```

Example:

Algorithm 5 executed on the following part of a tree:

Grand-parent	0010110000	$\to$	10010	$\to$	...
Parent	1010010000	$\to$	01010	$\to$	111
Child	0110010000	$\to$	10001	$\to$	001

Theorem 1.15 (Analysis of Algorithm 5). *Algorithm 5 terminates in $\log^* n$ time.*

Proof: A detailed proof is, e.g., in [Peleg 7.3]. In class we do a sketch of the proof.

Remarks:

- Colors $11*$ (in binary notation, i.e., 6 or 7 in decimal notation) will not be chosen, because the node will then do another round. This gives a total of 6 colors (i.e., colors 0,..., 5).
- Can one reduce the number of colors in only constant steps? Note that algorithm 3 does not work (since the degree of a node can be much higher than 6)! For fewer colors we need to have siblings monochromatic!
- Before we explore this problem we should probably have a second look at the end game of the algorithm, the UNTIL statement. Is this algorithm truly local?! Let's discuss!

Algorithm 6 Shift Down

```
1: Root chooses a new (different) color from $\{0, 1, 2\}$
2: **Each** other **node** $v$ concurrently executes the following code:
3: Recolor $v$ with the color of parent
```

Lemma 1.16 (Analysis of Algorithm 6). *Algorithm 6 preserves coloring legality; also siblings are monochromatic.*

Now Algorithm 3 (Reduce) can be used to reduce the number of used colors from six to three.

Algorithm 7 Six-2-Three

```
1: **Each node** $v$ concurrently executes the following code:
2: Run Algorithm 5 for $\log^* n$ rounds.
3: **for** $x = 5, 4, 3$ **do**
4:    Perform subroutine Shift down (Algorithm 6)
5:    **if** $c_v = x$ **then**
6:       choose new color $c_v \in \{0, 1, 2\}$ using subroutine **First Free** (Algorithm
         2)
7:    **end if**
8: **end for**
```

Theorem 1.17 (Analysis of Algorithm 7). *Algorithm 7 colors a tree with three colors in time $O(\log^* n)$.*

Remarks:

- The term $O()$ used in Theorem 1.15 is called "big O" and is often used in distributed computing. Roughly speaking, $O(f)$ means "in the order of f, ignoring constant factors and smaller additive terms." More formally, for two functions f and g, it holds that $f \in O(g)$ if there are constants x_0 and c so that $|f(x)| \le c|g(x)|$ for all $x \ge x_0$. For an elaborate discussion on the big O notation we refer to other introductory math or computer science classes.
- As one can easily prove, a fast tree-coloring with only 2 colors is more than exponentially more expensive than coloring with 3 colors. In a tree degenerated to a list, nodes far away need to figure out whether they are an even or odd number of hops away from each other in order to get a 2-coloring. To do that one has to send a message to these nodes. This costs time linear in the number of nodes.
- Also other lower bounds have been proved, e.g., any algorithm for 2-coloring the d-regular tree of radius r which runs in time at most $2r/3$ requires at least $\Omega(\sqrt{d})$ colors.
- The idea of this algorithm can be generalized, e.g., to a ring topology. Also a general graph with constant degree Δ can be colored with $\Delta + 1$ colors in $O(\log^* n)$ time. The idea is as follows: In each step, a node compares its label to each of its neighbors, constructing a logarithmic difference-tag

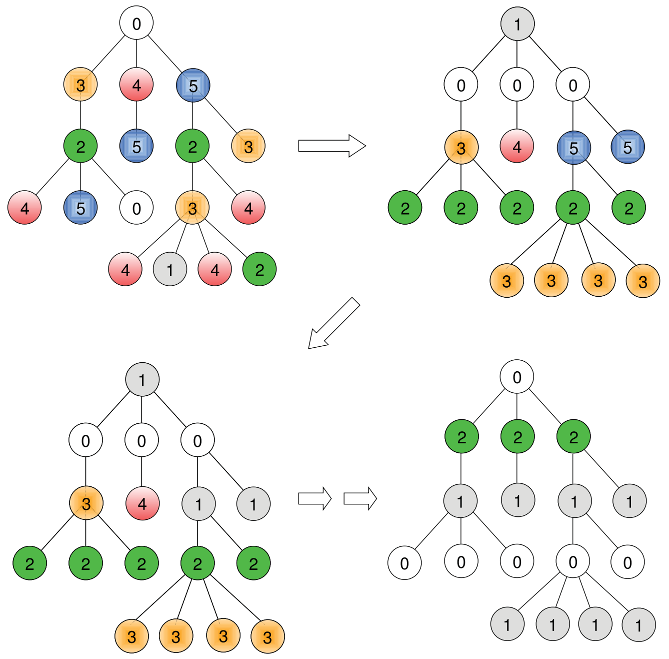

Figure 1.3: Possible execution of Algorithm 7.

as in 6-color (Algorithm 5). Then the new label is the concatenation of all the difference-tags. For constant degree Δ, this gives a 3Δ-label in $O(\log^* n)$ steps. Algorithm 3 then reduces the number of colors to $\Delta + 1$ in $2^{3\Delta}$ (this is still a constant for constant Δ!) steps.

- Recently, researchers have proposed other methods to break down long ID's for log-star algorithms. With these new techniques, one is able to solve other problems, e.g., a maximal independent set in bounded growth graphs in $O(\log^* n)$ time. These techniques go beyond the scope of this course.
- Unfortunately, coloring a general graph is not yet possible with this technique. We will see another technique for that in Chapter 10. With this technique it is possible to color a general graph with $\Delta + 1$ colors in $O(\log n)$ time.
- A lower bound by Linial shows that many of these log-star algorithms are asymptotically (up to constant factors) optimal. This lower bound uses an interesting technique. However, because of the one-topic-per-class policy we cannot look at it today.