Network Algorithms

Prof. Dr. Christian Scheideler

Technische Universität München, April 8 2008

1 Introduction

The goal of this course is to give an introduction to the state of the art in the theory of network algorithms. By "network algorithms" we mean algorithms for the design and management of logical networks (and their applications) as opposed to applications for physical networks like the Internet, though most of the applications we give in this course are actually for the Internet. Why are algorithms for the design and maintenance of logical networks (which we will also call *overlay networks*) important? Certainly, every distributed system must be based on some kind of logical interconnection structure allowing its sites to exchange information. Thus, in order to design efficient and scalable distributed systems, we first have to understand how to design efficient and scalable logical interconnection networks. Once a distributed system exceeds a certain size, one has to take into account that sites continuously enter and leave the system. This is because, for example, new sites may have to be added to maintain a certain service level or old sites may fail. Also, once a distributed system becomes very large, attacks on it by insiders and outsiders become more and more likely. Thus, besides addressing the problem of how to maintain an overlay network under a changing set of sites, one also has to address the problem of how to protect an overlay network against adversarial sites. We will investigate these issues in this course and present solutions that are based on state-of-the-art knowledge in this area. The first half of the course will focus on fundamental results in the area of networking, scheduling and data management, and the second half of the course will use these results to design efficient, scalable and robust overlay networks for particular applications including a shared file system and systems for anycasting, multicasting, and searching and sorting.

Before we start talking about algorithms and networks, we need a proper computational model, very much like mathematics needs axioms. It is a widely accepted fact that algorithmic advances in the area of computer science are only useful to society if they are based on models that truthfully reflect the restrictions and requirements of the corresponding applications. But what is a useful model for network algorithms, or distributed algorithms in general? We will start in this section with the discussion of properties such a paradigm should have, which is followed by the specification of a paradigm that we will use throughout the course. To understand the effects of this paradigm, a simulation environment will be provided so that algorithms can be implemented and tested within that paradigm. The only prerequisite to run this simulation environment is a PC that can compile C++ programs. For an introduction to C++ and C++ compilers see the web page of the course.

For the rest of this section, we first give some motivation and background information for the paradigm used in this course (Sections 1.1 and 1.2), and then we present the formal framework (Section 1.3) and the programming environment we will use in this course (Section 1.4). Finally, we give a simple example (Section 1.5) illustrating the use of this environment.

1.1 Towards a useful paradigm for distributed computing

Any paradigm that claims to be useful for distributed computing must be acceptable to all groups involved: users, developers, and scientists. This means that it has to satisfy three central demands:

- It must be *easy* to apply,
- it must allow the development of *efficient* and *robust* distributed programs, and
- it must be *simple and precise* to allow a verification and formal analysis of these programs.

Though in the academic world, ease of use may not be the most important issue, it should be clear that no matter how good a paradigm is, if it requires an expert to apply, it will not gain wide-spread acceptance. Also, a paradigm that does not allow the development of efficient distributed programs will most likely not be used for anything else than prototyping, and will therefore not make the transition from academia to industrial applications.

On the other hand, any programming paradigm that claims to allow the development of efficient and robust distributed programs must take the following issues into account:

- Sites operate in an asynchronous environment,
- sites may join and leave the system, or may simply fail,
- sites have different resources (processing cycles, memory, bandwidth), and
- messages have varying delays, or may simply get lost.

Thus, distributed programs should be given a high degree of freedom to manage their resources, which seems to forbid a paradigm which is easy to apply and precise. On the other hand, the freedom given to the developer should not be so high that it is tempting to produce inefficient code rather than efficient code. Thus, besides the paradox of achieving ease of use and preciseness together with a high degree of freedom at the same time, we also have to fight with the paradox of offering a high degree of freedom and restricting the development of inefficient code at the same time. Can there possibly be a paradigm that resolves these paradoxes?

Even if there is such a paradigm, would it be sufficient for distributed computing? Most probably not because besides simplicity and efficiency, robustness has become an increasingly pressing issue. Some attacks such as denial-of-service attacks cannot be solved in the algorithmic domain and therefore have to be addressed by a suitable paradigm. Also, basic security primitives such as secure communication channels should be built into the paradigm to prevent standard attacks like eavesdropping. However, approaches should be chosen here that do not endanger simplicity and efficiency. Is that possible?

1.2 Central demands

In order to investigate possible solutions, we need to structure our thoughts above and make them a bit more formal. What we are searching for is a *universal* paradigm that can address the following three central issues: *simplicity*, *efficiency* and *robustness*. Interestingly, these issues are highly dependent. Major challenges are to make the paradigm simple without losing universality, efficient without losing simplicity, robust without losing efficiency, and finally, universal without losing robustness. We discuss one by one the consequences of these requirements.

Simplicity

For a paradigm to be simple, it should be easy to *state*, *realize* and *apply*. The Turing machine, for example, does not satisfy these properties because although it is easy to state and realize, it is not easy to apply. A possible candidate for distributed computing could be a distributed version of the von Neumann machine: there are many processing units acting on a single shared memory. This is easy to state but not easy to realize because who should be responsible for maintaining the shared memory in a distributed environment? A more natural candidate is the way distributed tasks are often handled in real life: there are *subjects* with private, non-overlapping resources that exchange information. In this case, responsibilities are clear: every subject is responsible for the resources owned by it.

For a predictable execution of tasks within a subject, a subject should be an *atomic* entity residing at a *single, fixed* site. In order for the subject-based approach to be easy to apply, one has to take into account that concurrency *is* a difficult matter. Hence, concurrency should only happen between subjects but not within a subject. As a consequence, tasks should be executed by a subject in a strictly sequential manner, which implies that every execution of a task must be guaranteed to terminate in a finite amount of time. Since no finite time bound can be given for the interaction between subjects without losing the universality of the paradigm, this means that the execution of a task should not depend on the interaction with other subjects. Hence, no primitives should be allowed that require information from another subject for the execution of a task to proceed.

Another aspect for the subject-based approach to be easy to apply is that subjects should be immutable once created. That is, subjects cannot modify, add or delete variables or methods, though they can certainly modify the contents of their variables. This tremendously simplifies correctness proofs. Thus, if new variables or methods are needed, new subjects have to be spawned. If a subject A spawns a subject B, A is called the parent of B and B is called the child of A. For simplicity and transparency reasons, a child should be bound to the same site as its parent. Like in real life, a parent should be responsible for its child. In particular, a parent should be responsible for controlling the resources used by its child. In this way, resource responsibilities are well defined. This, in turn, implies that the parent relationship should not change because if it could, a parent may obtain the right to decide on the use of resources of a child at a remote site, which is not acceptable as it would introduce severe security and robustness risks.

Efficiency

For a paradigm to be efficient, no primitive should involve a large hidden overhead. Moreover, for simplicity and efficiency reasons, primitives should be selected so that the subjects are decoupled in space, time, and flow. *Space decoupling* means that the interacting subjects do not need to know their physical locations, *time decoupling* means that the interacting subjects do not need to be actively participating in an interaction at the same time, and *flow decoupling* means that the code execution inside subjects is not blocked by outside interactions. The space and time decoupling issues require an intermediate layer for the interaction between the subjects that can run concurrently with the subjects (in fact, we may treat it as another subject) that we will specify more precisely below.

Robustness

In order to allow the development of robust distributed algorithms, two central demands have to be met (see also [2, 3]):

1. **Owner consent and control:** All resources granted to a subject (such as time, space and bandwidth) should be under its control. For simplicity, it is best if subjects can only grant resources to their children. In this way, a subject only needs to control the resources of its children. Notice that these resources always belong to the same site since we do not allow subjects to migrate. Since, in addition, subjects cannot access anything outside of their realm directly, owner consent and control is assured.

2. **Minimal exposure:** The exposure of any information and resource due to the interaction with other subjects should be kept at a minimum, which includes the owner consent and control issue. Ideally, subjects should not be inspectable from outside and no information should be obtainable from a subject that can be used to take over its identity, even if the subject would want this. Only information that has been explicitly sent by the subject should be obtainable from that subject. Furthermore, a subject should be able to control the type and amount of information sent to it, which implies that a subject A should not be able to introduce a subject B to a subject C without B's consent. To minimize exposure in a parent-child relationship, initially there should only be a connection from the child to its parent, and not vice versa. This makes sure that subjects can, in principle, act independent of their environment so that subjects may just be verified once and then run anywhere with the same guaranteed outcome.

Simplicity is also important for robustness because it is a universal fact that every additional primitive increases the vulnerability of a paradigm. With respect to robustness, less is therefore more, though the universality and efficiency may suffer if this principle is exaggerated.

1.3 Formal framework

In order to establish a formal framework satisfying all of our demands above, we need to address two critical issues: primitives for a robust communication environment and primitives for a robust computational environment. The latter issue includes the problem of robust code migration and resource management.

Communication

We need the following ingredients to establish a robust communication infrastructure.

- subjects: active, static entities
- objects: passive, dynamic entities

- relay points: the only entities with identities

There are special objects called *clones* and *identities*. Clones and identities are so-called *dark* objects. That is, the information in them is not accessible to the subjects but only to the subjects environment. Let S denote the set of all subjects, I denote the set of all identities and R denote the set of all relay points. Identities can only be created for relay points but never for subjects.

Given a subject s, $p(s) \in S$ denotes the parent of s (i.e. the subject that created s). For an identity i, $s(i) \in R$ denotes the source of i (i.e., the relay point for which i was created), $d(i) \in R \cup \{\infty\}$ denotes the destination of i (i.e., the relay point i is meant for) and $b(i) \in R$ denotes the base of i (which we explain later in more detail). If $d(i) = \infty$, we call i a *public* identity and otherwise a *private* identity. Given a relay point r, $h(r) \in S$ denotes the home of r (i.e., the subject that created r) and $b(r) \in R$ denotes the base of r (to be explained later). $b()$, $s()$, $h()$ and $r()$ are not accessible to the subjects. Finally, let $E \subseteq R \times R$ denote the set of directed connections between the relay points.

Subjects, identities and relay points can be created or deleted. In the following, by $s.op(o_1 \mid o_2, o_3, \ldots)$ we mean that subject s applies method op to object o_1 using as parameters objects $o_2, o_3, \ldots$ First, we consider the case that a subject is created or deleted. When a subject s is created, it will have two relay points. Relay point $*_s$ is an internal relay point through which s receives tasks to be executed by s, but $*_s$ is not accessible by s, and relay point $\uparrow_s$ is the starting point of a connection from s to its parent. $\uparrow_s$ is accessible but cannot be deleted (without deleting s).

- s.create(s'): $S = S \cup \{s'\}$, $p(s') = s$, $R = R \cup \{*_{s'}, \uparrow_{s'}\}$, $h(*_{s'}) = b(*_{s'}) = s'$, $h(\uparrow_{s'}) = s'$, $b(\uparrow_{s'}) = *_s$ and $E = E \cup \{(\uparrow_{s'}, *_s)\}$.

- s.delete(s'): if $s = p(s')$ then $S = S \setminus \{s'\}$, $R = R \setminus \{r \mid h(r) = s'\}$, $E = E \setminus \{(r, r') \mid h(r) = s' \ \lor \ h(r') = s'\}$, and execute delete($s''$) for all $s'' \in S$ with $p(s'') = s'$.

Next, we consider the case that a relay point is created or deleted.

- s.create(r): $R = R \cup \{r\}$, $h(r) = s$, $b(r) = *_s$, and $E = E \cup \{(r, *_s)\}$.

- s.create($r \mid i$): if $h(d(i)) = s$ or $d(i) = \infty$ then $R = R \cup \{r\}$, $h(r) = s$, $b(r) = b(i)$, $E = E \cup \{(r, s(i))\}$ and $I = I \setminus \{i\}$ (i.e., the identity i is invalidated so that it cannot be used again).

- s.delete(r): $R = R \setminus \{r\}$ and $E = E \setminus \{(r', r'') \mid r' = r \ \lor \ r'' = r\}$.

Finally, we consider the case that an identity is created or deleted.

- s.create($i \mid r$): if $h(r) = s$ then $I = I \cup \{i\}$, $s(i) = r$, $d(i) = \infty$ and $b(i) = b(r)$.

- s.create($i \mid r, r'$): if $h(r) = s$ then $I = I \cup \{i\}$, $s(i) = r$, $d(i) = b(r')$ and $b(i) = b(r)$.

- s.delete(i): $I = I \setminus \{i\}$.

When looking carefully at these rules, the following important properties can be extracted:

- If a new subject is created, then initially there is only a link from that subject to its parent but not vice versa. In this way, the create operation can be implemented in a non-blocking way. Also, as we will see, all incoming information is under full control of the subject.

- A subject can only be deleted by the subject that created it. It cannot delete itself. Whenever a subject is deleted, also all of its descendants are deleted. In this way, there is always a connected tree of parent-child relationships.

- Any identity can only be used once. This limits the abuse of identities in order to start denial-of-service attacks, for example.

- Identities are necessary in order to establish links between relay points, but only to a relay point that created it. Thus, links can only be created to a subject by its explicit consent.

- An identity cannot be created for $*_s$. This makes sure that a subject can kill any connection to it at any time (by deleting either one of its relay points or a child subject).

- Relay points can establish linked lists. The destination of any such list is the base of all of its relay points. An identity created for any of the relay points in such a list is meant for the base of this list. In this way, lists can be shortcut. This is important to allow direct connections between any two subjects that may initially just be in the same connected component.

Due to the last two properties, we also call our approach *introduction by proxy, connection by base*.

Subjects communicate by exchanging objects along their connections. This is done by the following commands:

- s.verb(o): This generates a task for relay point $*_s$ to execute method "verb" with object o (o may also be NULL).

- $s.r$.verb(o): This generates a task for the base of relay point r of s to execute "verb" with object o (o may also be NULL).

Notice that for a robust *and* secure communication environment, communication links should be cryptographically secured so that they cannot be forged by anyone, and objects sent along a relay path should be encoded using the base of that path so that no intermediate node can read it. However, since in this course we will not address security issues, we will not discuss this issue further here.

Code migration

In order to allow the safe migration of subjects from one site to another, we use the concept of clones. Let C be the set of clones. For any clone c, let $s(c) \in S$ be the source of the clone and $d(c) \in S$ be the destination of the clone. A clone is created and deleted by the following operations:

- s.create(c): $C = C \cup \{c\}$, $c = s$, $s(c) = s$ and $d(c) = \infty$.

- s.create($c \mid r$): $C = C \cup \{c\}$, $s(c) = s$ and $d(c) = b(r)$

- s.create($s' \mid c$): if $c \in C$ and $d(c) \in \{s, \infty\}$ then execute s.create(s'), set $s' = c$ and $C = C \setminus \{c\}$ (i.e., c can only be used once).

A clone c *only* contains s itself, which means that c only contains the current state of the variables and methods in s as well as the requests that are currently queued in s but none of the relay points or connections established by s. Note that a clone can only be unwrapped once and only by the subject it is meant for.

For safe cloning, clones should be cryptographically secured so that they cannot be altered on a user level. It should only be possible to unwrap a clone by a secure platform within the site so that its code and data cannot be inspected or altered by the user. Though these are important issues, we will not address these issues further in this course since we want to focus more on robustness than security.

Resource management

Recall that the resources used by a subject should be under the control of its parent. We realize this with the help of the following operations:

- s.freeze(s'): if $s = p(s')$ then s' is frozen by s, which means that no requests will be executed for s' and its descendants.

- s.wakeup(s'): if $s = p(s')$ then s' is woken up by s, which means that now requests will again be executed by s' and its descendants (given that no ancestor of s gets frozen)

By default, a new subject is awake. A subject may also control which of its relay points is currently active. This is realized by the following operations:

- s.freeze(r): if $s = h(r)$ then r is frozen by s, which means that no requests will be processed (i.e., received and sent) by r.

- s.wakeup(r): if $s = h(r)$ then r is woken up by s, which means that now requests will again be processed by r.

By default, a new relay point is awake.

1.4 The subject-oriented programming framework

Now we are ready to describe our subject-oriented programming environment which will be used throughout the course. The basic ideas behind this framework date back to the actors model developed by Carl Hewitt at the MIT in the area of artificial intelligence [1], at a time when distributed computing was still in its infancy. His ideas have long been neglected and only recently saw a revival in programming languages such as E (see www.erights.org or [4]). However, all of the approaches based on his ideas only address efficiency and security issues but not robustness issues.

Layers of the framework

The subjects framework consists of three layers:

- **Network layer**: this is the lowest layer. It handles the exchange of messages between the sites.

- **Relay layer**: this handles the identity management and the exchange of messages between the subjects.

- **Subjects layer**: this is the layer for subject-oriented programs.

In the network layer, any given communication mechanism may be used, such as TCP/IP, Ethernet, or 802.11. Its management is entirely an internal matter of the relay layer. Hence, the relay layer allows to hide networking issues from the subjects so that subject-oriented programs can be written in a clean way. Thus, it remains to specify the subjects layer, the relay layer, and the interface between them.

The subject layer

All computation and storage in the subjects layer is organized into *subjects* and *objects*. A subject is an atomic thread with its own, private resources that are only accessible to the subject itself. "Atomic thread" means that a subject must be completely stored within a single site and that operations within a subject are executed in a strictly sequential, non-preemptive way. A prerequisite for this approach to work is that all elementary operations must be strictly non-blocking so that a subject will never freeze in the middle of a computation. A subject cannot access any of the resources outside of its private resources and the objects currently owned by it. The only way a subject can interact with the outside world is by sending objects to other subjects. A subject is bound to the site and the subject that created it.

The relay layer

All communication in the relay layer is handled via so-called *relay points*. Relay points can be thought of as ports but are much more general than that. A relay point is an atomic object that is bound to the subject that created it. The following demands have to be satisfied for the relay points:

- All calls to the same relay point are executed in FIFO order, i.e. in the order they were invoked by its subject.
- All objects sent from a relay point r to some relay point r' arrive at r' in the same order they were sent out by r (if they arrive).
- Objects are delivered in an at-most-once fashion. (Notice that exactly-once delivery cannot be guaranteed in a potentially unreliable network.)

Formal specification

There are three basic classes:

- **Subject**: base class for subjects.
- **Relay**: class for relay points, which are needed to interconnect the subjects.
- **Object**: base class for objects. There are two predefined subclasses of objects: **Identity** and **Clone**.

User-defined objects should be defined as follows:

```
class ⟨UserObject⟩: public Object
{
public:
```

```
  ⟨user-defined variables⟩
};
```

User-defined subjects should be defined as follows:

```
class ⟨UserSubject⟩:  public Subject
{
protected:
  ⟨user-defined variables⟩
  ⟨internal user-defined methods⟩

public:
  ⟨UserSubject⟩() and/or ⟨UserSubject⟩(⟨UserObject⟩ *o)
  {
    The only subject primitive that can be used here is "call".
  }

  ⟨callable user-defined methods⟩
};
```

All remotely callable user-defined methods must be of the form ”void ⟨method⟩()” or ”void ⟨method⟩ (⟨UserObject⟩ $*o$)”. More details can be found in the file Subjects1-4.h that can be downloaded from the course webpage (and might be given here later).

1.5 A simple example

In order to demonstrate our subjects framework, two examples can be found on the course webpage. See ”pingpong1-4.cpp” and ”intro1-4.cpp”. More details may be given here later.

References

[1] P. B. C. Hewitt and R. Stieger. A universal modular actor formalism for artificial intelligence. In *Proc. of the 1973 International Joint Conference on Artificial Intelligence*, pages 235–246, 1973.

[2] K. Cameron. The laws of identity. http://www.identityblog.com/stories/2004/12/09/thelaws.html, 2004.

[3] D. Epp. The eight rules of security. http://silverstr.ufies.org/blog/archives/000468.html, 2003.

[4] C. M. M.S. Miller and B. Frantz. Capability-based financial instruments. *Financial Cryptography 2000*, 2000. See also http://www.erights.org/elib/capability/ode/.