Theory of Network Communication

Dr. Christian Scheideler

Johns Hopkins University, Sept. 8 2004

1 Introduction

The goal of this lecture is to give an introduction to the state of the art in the theory of network communication. 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. This is certainly also true for network communication. For example, in general any infrastructure connecting processing units with each other may be called a network, but certain infrastructures such as a dedicated line between every pair of nodes are certainly unrealistic, because they are too expensive to build. Thus, messages may have to traverse several units to reach their destination, causing (among other problems) route selection and scheduling problems. Also, we will not just study network communication out of context, but we will also look at various topics that require or support efficient network communication such as distributed data management or design strategies for overlay networks. In fact, during the last two years the course has made a transition from a mere networking course to a course in which the principles behind efficient networking are used to develop efficient distributed algorithms and data structures. Therefore, it should come as no surprise that this lecture will mostly be concerned about distributed computing instead of just networks. Before going into details, let us first give an overview of this section. We start with some philosophical thoughts about distributed computing. They can be safely skipped if the reader is not interested in the motivation behind the distributed computing paradigm used in this course. Also, the views should not be seen as the ultimate truth but should be looked at from a critical angle. This is followed by an introduction to the spheres framework, a framework that we will use to argue about protocols and distributed algorithms and data structures. Finally, an introduction to the E language will be given, which will be used for implementations in the course.

1.1 Towards a useful paradigm for distributed computing

Any paradigm that claims to be useful for distributed computing must be acceptable by 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* 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 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 applications 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?

If so, we are searching for a paradigm that needs as few constructs as possible while offering a maximum amount of flexibility and avoiding major pitfalls in the development of inefficient code. To fulfill the latter two of these demands, a useful paradigm should

1. allow a distributed application to control the placement of processes and data as well as their degree of reliability,
2. introduce a hidden overhead that is as low as possible, and
3. enforce that processes in a distributed application are decoupled in space, time, and flow.

Space decoupling means that the interacting processes do not need to know their physical locations, *time decoupling* means that the interacting processes do not need to be actively participating in the interaction at the same time, and *flow decoupling* means that the code execution inside processes is not blocked by outside interactions.

Standard distributed computing approaches such as message passing, remote procedure calls, and shared spaces can only provide decoupling for a subset of these issues [2]. So a different approach is needed.

1.2 Why we should not view the network as a von Neumann machine

We all know how to write programs for a single computer. In doing so, we are usually following the von Neumann paradigm without being explicitly aware of it: Code and data are separate entities. The data is stored in a linear addressable and reliable memory and can be accessed at unit cost. So wouldn't it be a good idea to provide a platform for distributed applications that emulates the properties of a von Neumann machine, i.e. that provides a shared space and allows to establish processes acting on this shared space? The problem with this approach is that a universally applicable distributed shared memory cannot be implemented efficiently.

A major cause of this inefficiency is the communication overhead associated with managing the shared space in a scalable and reliable fashion. But even if these problems can be overcome by a shared memory environment, the mere attempt of hiding networking issues may create the illusion to the programmer that there is a single, natural design for a given application, regardless of the context in which that application will be deployed. This assumption can lead to the design of parallel programs that perform poorly in a distributed environment by using, for example, a parallelism that is too fine-grained to hide the network latency, or a communication pattern that does not reflect the underlying communication network. Other critical issues are:

1. Processes and data are *separate* entities. This creates access and sharing problems that have to be handled with great care to avoid inefficiencies and inconsistencies.

2. Read and write accesses can potentially *block*.

3. There is a *linear addressable* memory. This makes it tempting to use implicit and/or non-local strategies such as parallel hashing that are (or appear to be) efficient in theory (when using the unit-cost access assumption) but that are actually expensive to emulate in a distributed environment.

4. The memory is supposed to be *reliable*. This has made it tempting in the past to design distributed data structures in which only the *accesses* are parallelized but not the *structure* itself, creating data structures that are vulnerable in a distributed setting.

5. The memory management (mapping and access control) is *hidden* from the application. This prevents an application from optimizing the data placement to its needs.

6. It is not specified how processes can join an application. So this has to be handled by hand in distributed applications, making their design more complicated.

Finally, the shared memory concept can create security problems. In server-based systems, it may be possible for parts of a distributed application to run on top of a virtual address space without sacrificing memory protection: different parts of the application execute in private protection domains controlled by the server, and segments are shared only by mutual agreement. However, in open, server-less systems such as peer-to-peer systems, it may be impossible for the application platform to decide which part of a distributed application should have the exclusive right of using a particular memory segment, as long as the shared memory is a priori open to anyone.

The deficits above reveal that we have to make major changes if we want to arrive at a paradigm useful for distributed applications:

1. Instead of separating processes and data, organize all computation and storage into so-called *spheres*, autonomous, atomic threads with their own, private processing and data.

2. Instead of read and write operations, that may potentially block, only allow strictly non-blocking operations.

3. Instead of a linear addressable memory, there is no memory besides the private memory within spheres, and spheres are interconnected by an explicit link structure which is under the control of the application.
4. Instead of assuming reliable resources, spheres may fail.
5. Instead of a hidden resource management, the sphere placement and migration is under the control of the application.
6. Instead of a manual integration of new spheres, provide a rendezvous mechanism that allows spheres to find each other in an automatic way.

In the next section we present a formal framework that takes these insights into consideration, and we will argue why we believe that this framework is useful for theory and practice.

2 The Spheres Framework

In this section, we suggest a formal framework for the development of distributed applications. The basic ideas behind this framework date back to the actors model developed by Carl Hewitt at MIT in the area of artificial intelligence [1], at a time when distributed computing was still in its infancy. Unfortunately, software and hardware issues at that time prevented his ideas from becoming widespread. So people rather followed the von Neumann paradigm and its variants (including models such as BSP, logP, QSM, HMM, and many others).

2.1 The spheres framework

The spheres framework consists of three layers:

- **Network layer**: this is the lowest layer. It handles the exchange of messages between peers.
- **Contact layer**: this handles the exchange of messages between spheres and allows spheres to locate and interconnect to each other.
- **Spheres layer**: this is the layer for sphere-based applications.

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 contact layer. Hence, the contact layer allows to hide networking issues from the spheres so that sphere-based applications can be written in a clean way. Thus, it remains to specify the spheres layer, the contact layer, and the interface between them.

2.2 The spheres layer

All computation and storage in the spheres layer is organized into so-called *spheres*. A sphere is an atomic thread with its own, private resources that are only accessible by the sphere itself. "Atomic thread" means that a sphere must be completely stored within a single peer and that operations within a sphere 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 sphere will never freeze in the middle of a computation. A sphere cannot access any of the resources outside of its private resources. The only way a sphere can interact with the outside world is by sending messages to other spheres. A sphere is bound to the peer that created it. This is important to guarantee some basic properties for the exchange of messages between spheres specified below, but it does not restrict the universality of our approach.

2.3 The contact layer

All communication in the contact layer is handled via so-called *contact points*. Contact points can be thought of as ports but are much more general than that. A contact point is an atomic object that is bound to the sphere that created it. Given a sphere s, let $C(s)$ be the set of contact points created by it, and given a contact point c, let $R(c)$ be the set of spheres that have registered for it. Let *Contact* be the type of a contact point and *Reference* be the type of a far reference. Contact points are manipulated by the following operations:

- *Contact* **NewContact**(): this creates and returns a new contact point c to the sphere s calling the operation. Formally, $C(s) = C(s) \cup \{c\}$.
- **Reference**(*Contact* c): this returns a far reference to contact point c.
- void **Register**(*Contact* c): this allows a sphere s to register for contact point c. Formally, $R(c) = R(c) \cup \{s\}$.
- void **Unregister**(*Contact* c): this allows a sphere s to unregister for contact point c. Formally, $R(c) = R(c) \setminus \{s\}$.

We demand that **Register** and **Unregister** operations from the same sphere to the same contact point are executed in FIFO order, i.e. in the order they were called, so that eventually $R(c)$ arrives at a correct state.

A contact point c does not need a destructor. As long as $R(c) = \emptyset$, c will not be stored explicitly but only if some sphere registers for it. Variables of type *Contact* or *Reference* can be passed from one sphere to another. Ownership of a variable of type *Contact* allows to register and unregister for that contact point, and ownership of a variable of type *Reference* allows to send a message to the contact point. For those familiar with capability-based systems, adding unguessable authentication information to these types turns them into capabilities so that contact points are protected against unauthorized access.

Given a contact point c with far reference r, the "$\longleftarrow$" operator allows to send a message m to any sphere that has registered for c. Or more formally, if sphere s calls $\boldsymbol{r} \longleftarrow \boldsymbol{m}$, then the contact layer of s first checks whether s has already received a *representative* $s' \in R(c)$ from c. If not, s sends a request to c which returns any $s' \in R(c)$. Afterwards, s forwards the message to s'. If s' deregisters or becomes unavailable, the contact layer of s will request a new sphere from c.

The $\longleftarrow$ operation is a *non-blocking, eventual* send operator that guarantees the following properties:

- FIFO ordering: Messages sent by sphere s to some sphere s' via contact point c arrive at s' in the same order in which the $\boldsymbol{r} \longleftarrow \boldsymbol{m}$ operations were executed in s.

- At-least-once delivery: Messages are delivered to s' in an at least once fashion. (Notice that exactly-once delivery cannot be guaranteed in a potentially unreliable network.)

2.4 Creating and deleting spheres

Let $S(p)$ be the set of spheres that are currently at peer p. Spheres are created and deleted via the following operations.

- void **Spawn**($C(\cdot)$): this creates a new sphere s in the peer owning the sphere calling **Spawn**, using constructor $C(\cdot)$ for initializing s. Formally, $A(p) = A(p) \cup \{s\}$. Notice that no return value is provided. This makes sure that **Spawn** is non-blocking and the sphere calling **Spawn** has no initial access to s (which is useful for a formal security analysis).

- void **Halt**($D(\cdot)$): this deregisters the sphere s calling **Halt** from all contact points and waits until all tasks have been processed. (Notice that this happens in the background, i.e. the sphere will go on with executing code without waiting for the **Halt** operation to complete.) Afterwards, the destructor $D(\cdot)$ is called in s (which can be used, for example, to save the final state information of s or move it to another peer to recreate s there). Formally, $A(p) = A(p) \setminus \{s\}$.

For security reasons, the **Spawn** operation should only be allowed to be called by a special root sphere r which is created when the peer belonging to r initiates the platform providing the contact layer. In this way, the peer is under the control of which and how many spheres are spawned in it.

Notice that our model only requires seven primitives. All of these primitives are *necessary* in a sense that they cannot be replaced by a combination of the other operations. On the other hand, the primitives are also *sufficient* in a sense that they potentially allow a sphere to reside at any peer in the system and to communicate with any other sphere in the system. Thus, our framework is universal. Furthermore, it satisfies the demands we stated earlier on a distributed programming paradigm. Finally, notice its striking similarity with peer-to-peer systems: spheres are connected by a pointer structure in a similar way as peers are connected by an overlay network. In fact, the spheres framework allows to implement every peer-to-peer system that has been proposed so far, and much more.

Of course, one may wonder at this point whether the spheres framework will ever be more than just some thought experiment, but interestingly it is already much more than that. There is already a full-grown language support for it, called E. We will actually use E for implementations in this course.

2.5 A simple example

In order to demonstrate our framework, let us look at a simple example: We want to set up two nodes, Ping and Pong, that send messages back and forth. This can be done as shown in Figure 1. Root is the type of the root object, Ping is the type of the Ping node, and Pong is the type of the Pong node. We choose operations so that our algorithmic notation is as close as possible to the E language. (The example will be revised to an E program once we have set up a suitable platform in E providing the operations above.)

```
Root {
    Root() {
        $c_i$ := newContact()
        $c_o$ := newContact()
        Spawn(Ping($c_i$, Reference($c_o$)))
        Spawn(Pong($c_o$, Reference($c_i$)))
    }
}

Sphere Ping {
    PongRef: Reference

    Ping($c$ : Contact, $r$ : Reference) {
        Register($c$)
        PongRef := $r$
        PongRef $\leftarrow$ Output(”I am Ping”)
    }

    Output($s$ : String) {
        println(”Received ” $+s$)
        PongRef $\leftarrow$ Output(”I am Ping”)
    }
}

Sphere Pong {
    PingRef: Reference

    Pong($c$ : Contact, $r$ : Reference) {
        Register($c$)
        PingRef := $r$
    }

    Output(String $s$) {
        println(”Received ” $+s$)
        PingRef $\leftarrow$ Output(”I am Pong”)
    }
}
```

Figure 1: The Ping Pong example.

2.6 The E language

All necessary information about the E language can be found at www.erights.org. If you have problems with the language, the developer of the language, Mark Miller, can help you because he is attending the course.

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] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. To appear in *ACM Computing Surveys*, Microsoft Research, 2003.