Towards a Paradigm for Robust Distributed Algorithms and Data Structures

Christian Scheideler
Institut für Informatik
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany

April 3, 2006

Abstract

There is a wealth of literature on distributed algorithms and data structures. Standard models used in the research community are synchronous or asynchronous shared memory or network models. The shared memory model is basically a generalization of the von Neumann model from one processing unit to multiple processing units or processes acting on a single, linear addressable memory. In the network model, there is no shared memory. Every processing unit has its own, private memory, and the processing units are connected by a network of (usually) bidirectional communication links that allow the processing units to exchange messages. The set of processing units is usually considered to be fixed though processing units may fail and recover according to some stochastic or adversarial model.

With the rise of very large distributed systems such as peer-to-peer systems, these models are not appropriate any more. For example, the set of processing units can be highly dynamic and there may not be any mutual trust relationships between the units. This creates fundamental problems, such as keeping the (honest) units in a single connected component, that the previous models cannot address in their basic form. We show how to extend the network model so that we have a model that is powerful enough to design algorithms and data structures that are provably robust even against massive adversarial attacks. This model even allows to design strategies capable of addressing modern threats such as denial-of-service attacks and phishing that appear to lie outside of the algorithms domain.

1 Introduction

Large distributed systems are dynamic by nature because units may fail and new units may have to be added to include new resources or users. Once distributed systems become large enough, they will attract attacks and therefore need appropriate defense mechanisms to protect them against these attacks. Classical research on distributed algorithms and data structures has mostly ignored these issues and only focused on isolated distributed systems of a fixed size in which all units are honest (but may potentially fail). One of the motivations for this is that the problem of developing correct and efficient distributed algorithms and data structures for a static, honest system should be understood first before moving to dynamic systems. In fact, even developing correct and efficient distributed programs under ideal circumstances has turned out to be quite challenging. Once the problem is well-understood for a static system, extensions to dynamic systems may then be possible by migrating code and data in an appropriate way. Robustness against faults may be achieved by adding a certain level of redundancy, and robustness against adversarial behavior may be achieved with the help of techniques for secure multiparty computation.

However, not all attacks on a system can be handled with algorithmic techniques. Prominent examples here are denial-of-service attacks and phishing, i.e., attacks that either try to shut down certain parts of a system or to take over the identity of certain parts of a system in order to disrupt certain services or destroy the entire system. None of the models proposed for distributed computing in the theory community can address these attacks. Standard models are synchronous or asynchronous shared memory or network models. The shared memory model is basically a generalization of the von Neumann model from one processing unit to multiple processing units or processes acting on a single, linear addressable memory. In the network model, there is no shared memory. Every processing unit has its own, private memory, and the processing units are connected by a network of (usually) bidirectional communication links that allow the processing units to exchange messages.

In the (basic) shared memory model, every processing unit has the right to access any memory cell, and in the (basic) network model, a processing unit just has to hear about another processing unit in order to be able to send information to it. Both approaches introduce high security risks that makes them inappropriate for large distributed systems in an open environment, such as peer-to-peer systems. Therefore, a new approach is needed. In this paper, we first discuss the various issues such a new approach has to satisfy in order to allow the development of truly robust distributed programs, and then we propose a new paradigm that can address these issues in a satisfactory way. The paper ends with a conclusion.

2 Towards a paradigm for robust 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* 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 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?

2.1 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. Or more precisely, our programs usually consist of *active* (the processes) and *passive* (the data) entities. Data structures are mostly processed in an *exterior way*, i.e., by adjusting pointers into the data structure, rather than processing them in an *interior way*, i.e., let the objects in the data structure do the work. However, having active and passive entities creates access and sharing problems that have to be handled with great care to avoid inefficiencies and inconsistencies. This is further complicated by the fact that in many platforms and models, the passive entities are not under the control of the active entities. That is, some underlying layer may decide where passive entities may be placed and in which order requests may arrive at the passive entities. Although the approach of hiding the management of passive entities from the active entities was originally meant to *simplify* the design of distributed programs, it actually *complicates* it and produces inefficient programs. The inefficiency may be handled by developing programs with a course-grained parallelism (by following models such as BSP, logP, QSM, HMM, and many others), but doing it right often requires an expert. Finally, managing passive entities by some platform puts the burden of handling security and authorization issues on that platform. Active entities may not be able to adjust the security to their needs, unless the platform provides a suitable interface for that.

Hence, our basic approach will be *not* to distinguish between passive and active entities but to only allow active entities which we will call *subjects*.

2.2 Why distributed systems are hard to protect

Though a subject-based approach may help to simplify the design of distributed programs, would it also be able to address our efficiency concerns? Also, robustness issues need to be addressed because in recent years, robustness against adversarial behavior has become an increasingly pressing issue. Designing efficient and robust distributed systems is very challenging because of the following fundamental dilemma:

- *Efficiency* asks for *minimizing* the resources needed for the operations whereas
- *robustness* asks for *maximizing* the resources needed for an attack.

In cryptography, these seemingly contradicting requirements do not pose a problem because efficient cryptographic codes are known that are (believed to be) hard to break, such as RSA. However, in serverless systems like peer-to-peer systems, we cannot create such an asymmetric situation. For example, if a data item is only replicated among a few peers, then it does not matter how well it is protected by cryptographic techniques. A simple denial-of-service attack on all peers responsible for the data item will make it inaccessible. Even worse, with a relatively small investment of own resources, hackers can control a large pool of compromised resources over the Internet and use it to attack even powerful servers via so-called distributed denial-of-service attacks. Hence, in reality, the asymmetry is rather in favor of the attackers than in favor of the distributed system that needs to be protected. These attacks are certainly outside of the algorithmic domain and therefore need a new paradigm to protect against.

2.3 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. However, although it is easy to state, it is hard to realize and to apply in a distributed environment, as we have seen above. A more natural candidate is a subject-based approach: there are subjects with private, non-overlapping resources that exchange information.

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. As a consequence, 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. Notice that previous distributed computing approaches such as message passing, remote procedure calls, and shared spaces can only provide decoupling for a subset of these issues [8], which underlines the fact that a new approach is needed. Our approach of handling space and time decoupling is to use a light-weight intermediate layer for the interaction between the subjects that can run concurrently with the subjects (in fact, we may treat it as another subject). We will specify this layer (which we will call relay layer) more precisely below.

Robustness

In order to allow the development of robust distributed algorithms, three central demands have to be met (see also [5, 7]), which we also prefer to call the laws of robustness:

1. **User consent and control:** All resources of a site should be under its control. This means that also 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 directly outside of their realm, user consent and control is assured.

2. **Minimal exposure:** The secure platform should only provide the subjects with the minimal information necessary to perform their tasks and to maintain universality (i.e., so that all possible tasks can be implemented). This implies that subjects should not be inspectable from outside and therefore, only information that has been explicitly sent by the subject should be obtainable from that subject. If possible, no information should be obtainable by a subject that can be used to take over its identity, even if the subject would want this. 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.

3. **Minimal authority:** A subject should be given the minimum possible authority to execute any task. This can be ensured if a subject can only have direct access to its own data and a subject A can only connect to a subject B if B permits this. Hence, a subject A should not be able to introduce a subject B to a subject C without C's consent. The minimal authority condition is also known as the *principle of least privilege* [6, 11] or *principle of least authority* [10].

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.

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.

3.1 Communication

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

- subjects
- identities
- relay points
- secure links

Let S denote the set of all subjects, I denote the set of all identities, R denote the set of all relay points, and $E \subseteq R \times R$ denote the set of all secure links. 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 associated with i), $d(i) \in R \cup \{\infty\}$ 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 will 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).

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.

- s.create(s'): $S = S \cup \{s'\}$, $p(s') = s$, $R = R \cup \{*_{s'}, \downarrow_{s'}\}$, $h(*_{s'}) = s'$, $b(*_{s'}) = *_{s'}$, $h(\downarrow_{s'}) = s'$, $b(\downarrow_{s'}) = *_s$ and $E = E \cup \{(\downarrow_{s'}, *_s)\}$.
- s.delete(s'): if $s = p(s')$ then $S = S \setminus \{s'\}$, $R = R \setminus \{r \mid h(r) = s'\}$, $E = E \setminus \{(r, r') \mid h(r) = s' \ \vee \ h(r') = s'\}$, 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): $R = R \cup \{r\}$, $h(r) = s$, $b(r) = r$, and $E = E \cup \{(r, *_s)\}$.
- s.create($r \mid i$): if $h(d(i)) = s$ then $R = R \cup \{r\}$, $h(r) = s$, $b(r) = b(i)$, $d(i) = \infty$ and $E = E \cup \{(r, s(i))\}$.
- s.delete(r): $R = R \setminus \{r\}$ and $E = E \setminus \{(r', r'') \mid r' = r \ \vee \ r'' = r\}$.

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

- s.create(i): $I = I \cup \{i\}$, $s(i) = *_s$, $d(i) = \infty$ and $b(i) = *_s$.
- s.create($i \mid r$): if $h(r) = s$ and $r \neq *_s$ then $I = I \cup \{i\}$, $s(i) = r$, $d(i) = b(\downarrow_s)$ and $b(i) = b(r)$.
- s.create($i \mid r, i'$): if $h(r) = s$ and $r \neq *_s$ then $I = I \cup \{i\}$, $s(i) = r$, $d(i) = b(i')$ 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, the user consent and control requirement is satisfied because if a subject creates a new child, it *wants* a return value from that child since otherwise the child has no effect, but the child itself may not want to grant its parent permission to send it anything.

- A subject can only be deleted by the subject that created it. It cannot delete itself. In this way, parents have full control over their children.

- Whenever a subject is deleted, also all of its descendants are deleted. This is important to satisfy the user consent and control requirement since otherwise a parent may not be able to control the resources of its descendants.

- Only relay points are associated with identities, and a relay point can only have exactly one outgoing connection that is established when it is created. This is important to realize anonymity because different subjects must be accessed via different relay points having different identities, and therefore immediate associations with a single subject are not possible. Furthermore, a public identity does not need to contain any physical location information. In this case, even the relay layer of a subject does not know the physical location of a relay point when creating a private identity for it, so anonymity can be ensured in a very strong sense. Only the private identities have to store location information because otherwise connections cannot be established.

- Public identities can only be used to create private identities but not to create links between relay points. This ensures that an explicit permission must be given by a subject before another subject can connect to it. The reason why public identities are nevertheless necessary is to solve the initial contact problem because initially, subjects may not be connected, and so an offline process with public identities is necessary to connect them.

- Private identities can be used to establish links between relay points but only from a relay point of the subject it was meant for to the relay point representing its source. Since this source subject originally created the identity, this means that links can only be created by permission of the destination of the link. A private identity can only be used once to create a link.

- A private identity cannot be created for a $*_s$ point. 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*.

We notice that for a robust *and* secure communication environment, the communication links should be cryptographically secured so that they cannot be forged by anyone. Also, space decoupling has to be enforced, i.e., the subjects should not know the physical location of the other subjects they are communicating with but only some cryptographically secured identities (whose physical location information is only accessible to a protected relay layer) and handles to their own relay points.

3.2 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 $s(c) \in R$ be the source of the clone and $d(c) \in R$ 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 $d(c) = b(\downarrow_s)$.

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

- s.create($s' \mid c$): if $c \in C$ and $h(d(c)) = s$ then execute s.create(s'), set $s' = c$ and $C = C \setminus \{c\}$.

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 sphere 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 protected relay layer within the site so that its code and data cannot be inspected or altered by the user. This is important for digital rights management and secure grid computing.

3.3 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' (given that no ancestor of s gets frozen)

By default, a new subject is awake. The freeze and wakeup commands are very useful to help the platform decide when to move a subject to a storage device and when to move it back to local memory for execution.

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.

3.4 Further enhancements

Further enhancements of the subject-based paradigm are possible to widen the spectrum of applications it can be used for. For example, whenever a new relay point r is created with s.create(r), it is sometimes desirable to specify a policy for r. Possible policies are that r may only accept information up to a certain rate, r may only allow relay paths to it of some bounded length, or only specific methods in s can be called via r.

3.5 Message passing

Finally, we specify how to actually exchange information between the subjects. Messages can only be passed along links in E and the message passing is done with the help of the "←" operator. Two variants of this operator are possible:

- $s. \leftarrow m$: this sends message m to relay point $*_s$ so that it will be executed by s. (This is useful for s to stay alive or to produce a clone of itself that can wake up by itself when spawned by another subject.)

- $s.r \leftarrow m$ for some relay point r with $h(r) = s$: this sends a message m to the local relay point r which will then move it forward until the message arrives at some relay point $*_{s'}$ where it will be processed.

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

- FIFO ordering: all $\longleftarrow$-calls to the same relay point are executed in FIFO order, and messages sent by 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).
- At-most-once delivery: messages are delivered in an at most once fashion. (Notice that exactly-once delivery cannot be guaranteed in a potentially unreliable network.)

Similar concepts have also been used in the E language (www.erights.org).

4 The subject-oriented programming framework

Now we are ready to describe our subject-oriented programming environment which is already available as a simulation environment and used in the Network Algorithms course currently given at the Technical University of Munich [13]. 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 [9], at a time when distributed computing was still in its infancy. Unfortunately, software and hardware issues at that time prevented his ideas from becoming wide-spread in the distributed computing community. However, researchers in the programming language area have continued working on and extending Hewitt's ideas [1, 2, 3] which led, among other results, to the E language (see www.erights.org or [10]). The E language is probably the most advanced among these with respect to security as it uses a subject-based approach with cryptographically protected links, but it violates the laws of robustness that we formulated in Section 2.3.

4.1 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 and relay 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.

4.2 The subject layer

All computation and storage in the subjects layer is organized into subjects. 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. The only way a subject can interact with the outside world is by sending messages to other subjects. A subject is bound to the site and the subject that created it.

4.3 The relay layer

All communication between the spheres is handled by the relay layer. The relay layer manages the relay points as well as the identities of the relay points and the connections to other relay points. It also keeps track of the parent-child relationships between the subjects and unwraps clones. A relay point is an atomic object that is bound to the subject that created it. It has all properties of a subject except that it is not freely programmable but only supports the $\leftarrow$ operation. On the other hand, identities and clones are just cryptographically protected objects, i.e., they do not act themselves but are only used in certain actions.

4.4 Formal specification

There are four basic classes of objects:

- **Identity**: class for public and private identities
- **Relay**: class for relay points, which are needed to interconnect the subjects.
- **Clone**: class for clones, which are needed to migrate subjects safely from one site to another.
- **Subject**: base class for subjects.

Each of these basic classes offers a set of operations. If a subject s is executing operation *op*, we denote it as $s.op$, and if we want to specify the type of a certain parameter, we give the type in italic.

The identity class offers the following operations:

- s.new Identity: creates a public identity of subject s.
- s.new Identity(*Relay* r): creates a private identity of r for the parent of s.
- s.new Identity(*Relay* r, *Identity* i): creates a private identity of r for the base of identity i.
- s.delete i: this deletes Identity i

The relay class offers the following operations:

- s.new Relay: this creates a new relay point with a connection to s
- s.new Relay(i): this creates a new relay point r with a connection to the relay point identified by i, if i was meant for s.
- $s.r \leftarrow$ verb(args): if r has been created by s, a request to call verb(args) is sent to r.
- s.wakeup(r): s wakes up r if r is a relay point of s.
- s.freeze(r): s freezes r if r is a relay point of s.
- s.idle(r): returns true if r is idle (has no requests to process) and otherwise false.
- s.delete r: this deletes relay point r.

The clone class offers the following operations:

- s.new Clone: this creates a clone of s for the parent of s.

- s.new Clone(*Identity* i): this creates a clone of s for the base of i.

- s.delete Clone: this deletes a clone.

The subject class offers the following variables and operations:

- s.root: a handle to relay point $\downarrow_s$ which allows s to send requests to its parent.

- s.source: a positive number telling s the identity of the relay point that generated the request currently executed by s.

- s.sink: a positive number telling s the identity of the relay point in s that forwarded the request currently executed by s to s.

- s.new(*UserSubject*): spawns a new subject s' of type *UserSubject*. A subject pointer is returned to s allowing s to wake up, freeze, or kill s' but not to send requests to s'. By default, a new subject is awake. If a subject is frozen, no more requests are processed by it, and if a subject is killed, the subject and all of its descendants are removed instantly.

- s.new(Subject(*Clone* c)): spawns a new subject containing the clone in c if c is meant for s.

- s.delete(s'): this kills s' if s' is a child of s.

- $s. \leftarrow$ verb(args): a request to call verb(args) is created for s itself. This should be used when s is created so that s can become active, and it is useful for periodic control purposes.

- s.wakeup(s'): s wakes up s' if s' is a child of s.

- s.freeze(s'): s freezes s' if s' is a child of s.

- s.idle(): checks if there are still messages for s.

4.5 A simple example

In order to demonstrate our subjects 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. The main() function starts the ping-pong process by creating the Pong object.

5 Robustness properties

At this point, the question may certainly arise whether the subjects framework can really protect a system against denial-of-service attacks and identity theft. In this section, we will discuss how the subjects framework might be implemented so that a protection can indeed be achieved.

5.1 Denial-of-service attacks

The problem with denial-of-service attacks is that it is difficult to defend against them at the destination of the attack. Hence, it would be desirable to prevent them at the source. Our subjects framework can achieve this without changing the core of the Internet. Only the edge of the Internet, i.e., the place where the Internet service providers (ISPs) interact with their clients, needs to be changed, which is possible since

```
Subject Ping {
    $num$: integer
    $ponglink$: Relay

    Ping() {
        num := 0
        ← Init()
    }

    Init() {
        $i$: Identity

        $ponglink := $ new Relay
        $i := $ new Identity($ponglink$)
        $root \leftarrow$ Setup($i$)
        PingPong()
    }

    PingPong() {
        if ($num < 5$) then
            $num := num + 1$
            $root \leftarrow$ PingPong()
    }
}

Subject Pong {
    $pinglink$: Relay

    Pong() {
        ← Init()
    }

    Init() {
        new(Ping)
    }

    Setup($i$: Identity) {
        $pinglink := $ new Relay($i$)
        delete $i$
    }

    PingPong() {
        $pinglink \leftarrow$ PingPong()
    }
}

main() {
    Subject $pong := $ new(Pong);
    run($pong$,40)
}
```

Figure 1: The Ping Pong example.

no standards are needed for this. Suppose that the relay points are managed by the ISP whereas all other aspects are managed by the clients. Given that the ISPs are honest, a client would not be able to create relay points at will and therefore would not be able to connect to any other client as it likes. An authorization process through identities could be enforced by the ISPs so that a user A can only connect to a user B if B explicitly allows that. Hence, unauthorized transmissions of messages can be prevented at the source. Suppose, instead, that client A has a correctly authorized connection to client B and wants to use it now for a denial-of-service attack. Then B can defend itself against it by simply killing the local relay point through which A is connected to B. In this way, A's authorization will be revoked and A will not be able any more to send messages to B. Hence, as long as the honest users form a single connected component, denial-of-service attacks can, in principle, be prevented without losing any of the honest users. Of course, one still has to solve the problem of how to keep the honest users in a single connected component under adversarial attacks, but fortunately, very simple strategies have just recently been proposed that can solve this problem with a low overhead [4, 12].

5.2 Identity theft

Also identity theft can be prevented to a certain degree. Suppose that all the relay layer lets the subjects know about their relay points are local handles, and whenever a request is processed by a subject, it only learns about a one-way hash value of the identity of the source and sink of the request. Furthermore, given a critical connection from A to B that needs to be protected, B only accepts a change of this connection in an offline mode. Then we want to show that identity theft via online attacks are not possible.

For this, consider the situation that subject A has a connection to subject B, and this connection starts at relay point r in A and ends at relay point r' in B. Suppose there is another subject C that wants to take over the identity of A, i.e., that wants to transmit messages to B so that B thinks they are from A (which it only does if the source of the message indicates r and the sink of the message indicates r'). Further, suppose that A and B are naive and deliver any information that C requests. Since A and B do not have access to the true identities of r and r' but only their one-way hash values, C would not be able to recover the true identities. C could still ask A to establish a connection to r or ask B to establish a connection to r', but that would not allow C to give B the impression that A is talking to B. Also, C cannot ask A to bend its connection from r to r' over to a relay point in C because once the outgoing connection of a relay point has been set, it cannot be changed any more. The only way A could establish a connection to C is to use a new relay point, but this relay point has an identity that is different from r, and therefore B would not accept messages from A via that new relay point as coming from A. Thus, no online strategy is available for C to take over A's identity.

6 Conclusions

In this paper we demonstrated that it is possible to design a paradigm for distributed computing that can address all major demands for simplicity, efficiency and robustness. We hope that this paradigm will create some interesting discussion that will ultimately lead to a new, much safer version of the Internet as it is today. Also, we hope that our paradigm, or an appropriate enhancement of it, will serve as a base for fundamental research on provably efficient and robust distributed algorithms and data structures in the future. The author certainly welcomes any comments and suggestions.

7 Acknowledgements

I would like to thank Mark Miller for many helpful discussions and for introducing me to his E language.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[2] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, 1997.

[3] G. Agha and T. Prasanna. *Formal Methods for Distributed Processing - An Object Oriented Approach*, chapter 8, Actors: A model for reasoning about open distributed systems. Cambridge University Press, 2001.

[4] B. Awerbuch and C. Scheideler. How to spread peers in virtual space. Unpublished manuscript, available on request, November 2005.

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

[6] P.J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, 1976.

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

[8] 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.

[9] C. Hewitt, P. Bishop, 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.

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

[11] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[12] C. Scheideler. How to spread adversarial nodes? Rotate! In *Proc. of the 37th ACM Symp. on Theory of Computing (STOC)*, pages 704–713, 2005.

[13] C. Scheideler. Network algorithms. See http://www14.in.tum.de/lehre/2005WS/na/index.html.en, 2005.