



PEPITO
IST-2001-33234

PEer-to-Peer Implementation and TheOry

Deliverable no: D1.2

Survey of formal challenges and solutions for peer-to-peer computation

REPORT VERSION: first

REPORT PREPARATION DATE: 2005.2.28

CLASSIFICATION: Public

DELIVERABLE NO: D1.2 DUE DATE: Month 38 DELIVERY DATE: Month 38

PROJECT START DATE: 2002.01.01 PROJECT DURATION: 36+4 months

RESPONSIBLE PARTNER: UCAM

PARTICIPATING PARTNERS: EPFL, INRIA, KTH, UCAM, UCL

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge UK



Project funded by the European Community under the 'In-formation Society Technologies' Programme (1998–2002)

Project Number: IST-2001-33234

Project Acronym: PEPITO

Title: PEer-to-Peer Implementation and TheOry

Deliverable No: D1.2

Survey of formal challenges and solutions for peer-to-peer
computation

Due date: project month 38

Delivery date: 2005.2.28

Responsible Partner: UCAM

Participating Partners: EPFL, INRIA, KTH, UCAM, UCL

9th March 2005

Editor: Peter Sewell

Authors: Raphaël Collet, Vincent Cremet, Kevin Glynn, Yves Jaradin, Valentin Mesaros, Uwe Nestmann, Peter Sewell, Peter Van Roy.

Contents

1 Overview	4
1.1 Task: Transactions and the Semantics of Failure	4
1.2 Task: Models of Peer-to-Peer Group Collaboration	4
2 Transactions and the Semantics of Failure	5
2.1 Low-level failure semantics	5
2.2 Unreliable Failure Detectors	5
2.3 Transaction Axiomatization	5
2.4 A Transactional System for Structured Overlay Networks	6
2.4.1 Motivation and Contribution	6
2.4.2 Principles	7
2.4.3 Architecture	7
2.4.4 Dealing with Failures and Network Changes	8
3 Models of Peer-to-Peer Group Collaboration	8
3.1 Verifying DKS: The Static Case	8
3.1.1 Introduction	9
3.1.2 Why DKS?	9
3.1.3 Verification approach	9
3.1.4 Contributions	10
3.1.5 Related Work	10
3.1.6 Future Work	10
3.1.7 Conclusions	11
3.2 Reasoning about executable distributed code	11
3.3 Integration: executable Acute code for Chord, Nomadic Pict, and Ambients	11
3.4 Models for anonymity analysis	12
3.5 Authority confinement by membranes	12
3.5.1 Problem context	12
3.5.2 Related work	12
3.5.3 Formal calculi and language design work	13
3.5.4 Membrane design guidelines	13

3.5.5	Overview of membrane concepts	13
3.5.6	Future plans	14
3.5.7	Conclusion	14
3.5.8	Acknowledgement	15
4	Other work	15
5	Relations to other work packages	15
6	Conclusion	16

1 Overview

In Deliverable D1.1 *Required foundations for peer-to-peer systems* we discussed the requirements for satisfactory formal foundations of P2P systems. We highlighted five areas:

- ▷ Specification of subtle distributed algorithms
- ▷ Verification
- ▷ Fault tolerance
- ▷ Mobility
- ▷ Programming language design

We have addressed all five of these. The last two are now reported on in WP3, e.g. with the Acute language design work supporting computation mobility (but see §3.3 below for libraries supporting higher-level mobility combinators).

For the first three, work has proceeded at three levels of abstraction, spanning many of the critical features of real-world P2P systems. There are two interlinked tasks:

1.1 Task: Transactions and the Semantics of Failure

1. At the lowest level we have built a precise semantic model of failures as they occur in distributed communication, specifying the behaviour of the TCP and UDP protocols, with their Sockets API library interface.
2. These low-level protocols do not provide good failure detection to applications. We have worked on process-calculus models of Chandra-Toueg style failure detectors. (Related work on a model implementation of such failure detectors, above our specified TCP/UDP layer, was described in WP2).
3. At the highest level, P2P systems must operate correctly in the presence of failure. This may be achieved through provision of transactional idioms. Work has proceeded on three lines: on an axiomatisation of desired transactional properties, on a model OCaml implementation of distributed transactions, and on a transactional system for structured overlay networks.

1.2 Task: Models of Peer-to-Peer Group Collaboration

In this task we developed specific models and reasoning techniques focussing on different aspects of P2P systems:

- ▷ Verifying DKS: The static case
- ▷ Reasoning about executable distributed code
- ▷ Integration: executable Acute code for Chord, Nomadic Pict, and Ambients
- ▷ Models for anonymity analysis
- ▷ Authority confinement by membranes

2 Transactions and the Semantics of Failure

2.1 Low-level failure semantics

A main goal here was to establish an accurate failure semantics for low-level interaction. As we stated in the project proposal:

“We will develop rigorous semantic models of the low-level interactions and failures that may occur in a peer-to-peer system — not in the rather abstract styles of traditional distributed algorithm theory or process calculi, but for the actual networking APIs used.”

A description of our initial UDP specification was published during the first year of the project [45, 28], and in the final period of PEPITO the full goal has been achieved. We have produced a behavioural specification of the networking *Sockets API* and the underlying TCP and UDP protocols that exactly characterises behaviour in the presence of message loss, malicious attack with arbitrary messages, and arbitrary use and misuse of the API. It is mathematically rigorous, detailed, and has been shown to be accurate by an experimental validation process. The specification and an extended discussion document have been made available; one paper has been submitted for publication and more will follow in the future [5, 6, 7]. The work establishes practical techniques for rigorous description and automated testing that can also be used for future protocol designs, at either the transport or peer-to-peer overlay levels.

We refer to the reader to Deliverable D1.4 *Study of possible semantics of failures* for further details of this work.

2.2 Unreliable Failure Detectors

The concept of unreliable failure detectors for reliable distributed systems was introduced by Chandra and Toueg as a fine-grained means to add weak forms of synchrony into asynchronous distributed systems. Various kinds of such failure detectors have been identified as each being the weakest to solve some specific distributed programming problem. In [26], we provide a fresh look at failure detectors from the point of view of programming languages, more precisely using the formal tool of operational semantics. Inspired by this, we propose a new failure detector model that we consider easier to understand, easier to work with and more natural. Using operational semantics, we prove formally that representations of failure detectors in the new model are equivalent to their original representations within the model used by Chandra and Toueg. In summary, we provide an original presentation, using operational semantics, of existing work by Chandra and Toueg, which is targeted at an audience in process calculi and programming language semantics. As main contribution, the new model to represent failure detectors eliminates a number of drawbacks of the original model used by Chandra and Toueg. Many other failure detectors have been studied in the literature; up to now, we restricted our attention to the ones introduced by Chandra and Toueg and by Chandra, Hadzilacos and Toueg.

2.3 Transaction Axiomatization

In the second year we completed our work on the axiomatization of transactions. The deliverable D1.3, available since June 2003, is our final report for the project. This work also led to a conference publication [8].

Fault-tolerance is central to the design of peer-to-peer systems. Transactions are a common idiom for programming in the presence of failure, assisting in the design of robust and fault-tolerant distributed systems. In this work we constructed a formal model of this aspect of distributed computation.

Transactions are commonly described as being ACID: Atomic, Consistent, Isolated and Durable. However, although these words convey a powerful intuition, the ACID properties have never been given a precise semantics in a way that disentangles each property from the others. Among the benefits of such a semantics would be the ability to trade-off the value of a property against the cost of its implementation.

Our contribution is to give a sound equational semantics for the transaction properties. We define three categories of actions, A-actions, I-actions and D-actions, while we view Consistency as an induction rule that enables us to derive system-wide consistency from local consistency. The three kinds of action can be nested, leading to different forms of transactions, each with a well-defined semantics. Conventional transactions are then simply obtained as ADI-actions.

From the equational semantics we develop a formal proof principle for transactional programs, from which we derive the induction rule for Consistency.

We hope that this theoretical work on the decomposition of transactions can be applied to the design and implementation of simple and well-understood language constructs for transactional computation in a peer-to-peer context. Contrary to the domain of databases where a transaction is always supposed to satisfy the four ACID properties, using transactions at the level of applications can lead to choose only a subset of these properties to match specific requirements, for instance in terms of efficiency.

2.4 A Transactional System for Structured Overlay Networks

2.4.1 Motivation and Contribution

We are interested in running transactions in a dynamic decentralized system, where nodes could fail with a relatively high rate, and where data items could be temporary inaccessible. To our knowledge, currently there exists no system working in such a dynamic environment that provides all the ACID properties of transactions. The only systems that provide ACID properties are more or less centralized, and limited in the number of participating sites. This is, for instance, the case of the GlobalStore [2] fault tolerant system. Although it provides rich functionality for doing transactions on objects, the lock management is centralized.

On the other hand, decentralized systems working in dynamic environments provide only some of the ACID properties, or most of them only to some extent. Some examples of such systems include the following. The CFS [41] system provides read-only storage. Ivy [25] is a multi-user read/write peer-to-peer file system, with no centralized or dedicated components. However, it does not guarantee write-read consistency, and it is suitable only for small groups of cooperating participants.

In the present paper [23] we extend a lightweight transaction protocol and provide an architecture implementing a robust, full transactional system on top of structured overlay networks. The system can be implemented as a service that runs on top of P2PKit [17, 18]. The latter provides a platform for services, and uses our P2PS [22, 11] library for its underlying P2P network.

Our transactional system can be seen as a lightweight group management protocol. Given a large set of objects distributed over an overlay network, a transaction consists of a (usually small) dynamic number of objects that are temporarily collaborating to perform the transaction. The transactional protocol provides an atomic multicast with rollback capability for any number of such temporary groups concurrently. We therefore see our transactional system as a natural service for applications written on top of decentralized overlay networks.

2.4.2 Principles

In our system the data items have the form of objects. The objects are spread over nodes that form a peer-to-peer network. Given the scalability properties of the P2P system, the procedures for storing and accessing the objects within the system will eventually respect load-balancing criteria. As the P2P system behaves like a distributed hash table (DHT), putting an object in the system implies storing it at the node responsible for the corresponding hash value. All accesses to an object will go through the responsible node and will be controlled by the transaction service local to that node.

We use strict two-phase locking to guarantee the serializability of transactions. In order to avoid deadlocks, we assign priorities to transactions. Older transactions have priority over newer ones. When a transaction waits for a lock that is held by a transaction with less priority, the system prevents the latter transaction from acquiring new locks. The low priority transaction is said to be *on probation*. This transaction must therefore either terminate (commit or abort), or restart with the same priority. Restarting the transaction implies to release all its locks, which gives the other transaction the opportunity to complete. This algorithm is taken from [44]. We have adapted it to make it distributed. We note that priorities can be assigned in a decentralized manner by using an approximate global clock. No centralized priority assignment algorithm or sophisticated clock synchronization algorithm is needed, since the synchronization only needs to be approximate.

Note that we consider the persistence of data an orthogonal issue. One can integrate a concurrent service on the peer-to-peer network that saves consistent snapshots of the objects. It is even possible to implement it as a transaction.

2.4.3 Architecture

We propose an architecture that follows the principles mentioned above. We make a separation of concerns, and implement a transaction with two kinds of processes: a transaction manager (TM), and transaction participants (TP). Both the transaction managers and participants interact with transaction service nodes (TS).

- ▷ Each transaction has one **transaction manager**. The TM is located at the site where the transaction is initiated. The transaction manager executes the actual transaction procedure, and manages the transaction, taking the decision to eventually commit, abort, or restart the transaction. The TM does not directly access the shared data, nor does it manage the associated locks and the related conflicts. It interacts with the transaction participants related to the objects involved within the transaction.
- ▷ A **transaction participant** is the local representative of a transaction on an overlay node. Each overlay node involved in a transaction has a TP for that transaction. The transaction participant is responsible for acquiring locks to access the shared objects, and invokes methods on those objects. It returns object invocation results, and notifies probation to its transaction manager.
- ▷ Each node in our overlay network has a **transaction service**. The TS creates the local TM and TPs involved in a transaction, and manages the access to shared data. In our model, the TS manages the priority queues and the related locks associated to objects on its overlay node.

The following **consistency invariant** must be ensured.

If a transaction participant commits (resp. aborts, restarts), then all the participants of the transaction that are not failed eventually commit (resp. abort, restart).

This invariant guarantees the consistency of the objects' states. It requires that the communication and lookup in the overlay network must be reliable. But it can be maintained with simple techniques, which keep the architecture scalable.

2.4.4 Dealing with Failures and Network Changes

As we said above, we simply have to ensure the consistency invariant. Thus, only the failures of the overlay nodes involved in a transaction have to be treated. Our system deals with node failures (transaction manager and participants), and with nodes joining and leaving.

Consider the failure of a transaction manager's node. This can be handled by maintaining a lightweight replica of the TM. The replica only needs to know the participants of the transaction, and the decision taken by the TM (commit, abort, or restart). If the failure happens after the TM has taken a decision, its replica can take over and complete. Otherwise, the replica will restart the transaction.

As another example, suppose an overlay node wants to leave the network, and an object on this node is locked by a transaction. This situation can be handled gracefully, by putting the local transaction participant on probation. The TP will interact with its TM to either complete the transaction, or to restart it after the node has left.

3 Models of Peer-to-Peer Group Collaboration

3.1 Verifying DKS: The Static Case

Abstract

Structured peer-to-peer overlay networks can be seen as a class of algorithms that provide efficient message routing for distributed applications using a sparsely connected communication network. In this paper, we formally verify a typical application running on a fixed set of nodes. This work is intended as a foundation for studies of more dynamic systems.

We identify a value and expression language for a value-passing CCS (*Calculus for Communicating Systems*) that allows us to formally model a distributed hash table implemented over a static DKS overlay network. We then provide a specification of the lookup operation in the same language, allowing us to formally verify the correctness of the system in terms of observational equivalence between implementation and specification. For the proof, we employ an abstract notation for reachable states that allows us to work conveniently up to structural congruence, thus drastically reducing the number and shape of states to consider. The structure and techniques of the correctness proof are reusable for other overlay networks.

Recently, we found that our proof techniques can be interpreted via an extension of the so-called *Cones and Foci* framework [14], so this verification task also feeds nicely back into the Concurrency Theory community. We are currently working on a journal version where this feedback is explained abstractly and then applied to the static DKS algorithm, but it will not be finished during the course of the PEPITO funding period.

3.1.1 Introduction

Although in recent years, decentralised structured peer-to-peer (p2p) overlay networks [29, 42, 33, 32] have emerged as a suitable infrastructure for scalable and robust Internet applications, to our knowledge, no such system has been formally verified.

One commonly studied application is a *distributed hash table* (DHT), which usually supports at least two operations: the insertion of a (key,value)-pair and the lookup of the value associated to a given key. For a large p2p system (millions of nodes), careful design is needed to ensure the correctness and efficiency of these operations, both in the number of messages sent and the expected delay, counted in message hops. Moreover, the sheer number of nodes requires a sparse (but adaptable) overlay network.

3.1.2 Why DKS?

DKS, as developed by PEPITO-partners and described in the respective project deliverables, builds upon the idea of *relative division* [30] of the virtual space, which makes each participant the root of a virtual spanning tree of logarithmic depth in the number of nodes.

In addition to *key-based routing* to a single node, which allows implementation of the DHT interface mentioned above, the DKS system also offers key-based routing either to all nodes in the system or to the members of a multicast group. The basic technique used for maintaining the overlay network, *correction-on-use*, significantly reduces the bandwidth consumption compared to its earlier relatives such as Chord [42], Pastry [33] and Can [32].

Given these features, we consider the DKS system as a good candidate infrastructure for building novel large-scale and robust Internet applications in which participating nodes share computing resources as equals. We thus also consider it a valuable example to exercise formal verification techniques.

3.1.3 Verification approach

In our work, we report on the first results of our ongoing efforts to formally verify DHT algorithms. We initially focus on *static* versions of the DKS system: (1) they comprise a fixed number of participating nodes; (2) each node has access to perfectly accurate routing information. As a matter of fact, already for static systems formal arguments about their correctness turn out to be non-trivial.

We consider the correctness of the *lookup* operation, because this operation is the most important one of a hash table: under all circumstances, the data stored in a hash table must be properly returned when asked for. (The insert operation is simpler to verify: the routing is the same as for lookup, but no reply to the client is required.)

We analyse the correctness of lookup by following a tradition in *process algebra*, according to which a reactive system may be formulated in two ways. Assuming a suitably expressive process calculus at our disposal, we may on the one hand *specify* the DHT as a very simple purely sequential monolithic process, where every (lookup) request immediately triggers the proper answer by the system. On the other hand, we may *implement* the DHT as a composition of concurrent processes—one process per node—where client requests trigger internal messages that are routed between the nodes according to the DKS algorithm. The process algebra tradition says that if we cannot distinguish—with respect to some sensible notion of equivalence—between the specification and the implementation regarded as black-boxes from a client’s point of view, then the implementation is correct with respect to the specification.

3.1.4 Contributions

While the verification follows the general approach mentioned above, we find the following individual contributions worth mentioning explicitly.

1. We identify an appropriate expression and value language to describe the virtual identifier space, routing tables, and operations on them.
2. We fix an asynchronous value-passing process calculus orthogonal to this value language and give an operational semantics for it.
3. We model both a specification and an implementation of a static DKS-based DHT in this setting.
4. We formally prove their equivalence using weak bisimulation. In detail:
 - ▷ We formalise transition graphs up to structural congruence.
 - ▷ We develop a suitable proof technique for weak bisimulation.
 - ▷ We design an abstract high-level notation for states that allows us to succinctly capture the transition graphs of both the implementation and the specification up to structural congruence.
 - ▷ We establish functions that concisely relate the various states of specification and implementation.
 - ▷ We show normalisation of all reachable states of the implementation in order to establish the sought bisimulation.

The work is published in [10], while the formal proofs are found in the long version of the paper [9].

3.1.5 Related Work

To our knowledge, no peer-to-peer overlay network has yet been formally verified. That said, papers describing such algorithms often include pseudo-formal reasoning to support correctness and performance claims.

Previous work in using process calculi to verify non-trivial distributed algorithms includes, e.g., the two-phase commit protocol [3], a sliding-window protocol [15] and a fault-tolerant consensus protocol [27]. However, in these algorithms, in contrast to overlay networks, each process communicates directly with every other process.

Other formal approaches, for instance *I/O*-automata [21] have been used to verify traditional (i.e., logically fully connected) distributed systems; we are not aware, though, of any p2p-examples.

3.1.6 Future Work

Peer-to-peer algorithms in general are likely to operate in environments with high dynamism, i.e., frequent joins, departures and failures of participating nodes. This case gives us increased complexity in three different dimensions: a more expressive model, bigger algorithms and more complex invariants.

To cope with dynamism, structured peer-to-peer overlay networks are designed to be stabilising. That is, if ever the dynamism within the system ceases, the system should converge to a *legitimate* configuration. Proving, formally, that such a property is satisfied by a given system is a challenge that we are currently addressing in our effort to verify peer-to-peer algorithms.

The work present in this paper is a necessary foundation for the more challenging task of formal verification of the DKS system in a dynamic environment.

3.1.7 Conclusions

The use of process calculi lets us verify executable formal models of protocols, syntactically close to their descriptions in pseudo-code. We demonstrate this by verifying the DKS lookup algorithm. Our choice to work with a reasonably standard process calculus, rather than the pseudo-code that these algorithms are expressed in, made it only slightly harder to ensure that the model corresponded to the actual algorithm but let us use well-known proof techniques, reducing the total amount of work.

Other overlay networks, like the above-mentioned relatives of DKS, would require changes to the expression language of the calculus as well as the details of the correspondence proof; however, we strongly conjecture that the structure of the proof would remain the same.

3.2 Reasoning about executable distributed code

Work such as that in §2.1 is needed to characterise the existing failure semantics, but it is also necessary to establish practical techniques for reasoning about systems above that semantics. In particular, we aimed to make it possible to reason about *executable* descriptions of distributed algorithms, rather than idealised pseudocode or automata, in a fully formal manner. We have demonstrated that this is possible, reasoning within the Isabelle proof assistant about programs written in an executable language (a fragment both of OCaml and of our Acute language developed in WP3) above our earlier UDP/Sockets semantics. Fully-formal reasoning about any executable code is very challenging, so we have attempted only modest examples to date, not a full-fledged P2P algorithm, but the paper [12] addresses many of the key difficulties.

3.3 Integration: executable Acute code for Chord, Nomadic Pict, and Ambients

As exercises in integration, we wrote three communication libraries in our Acute language (of Workpackage 3) above the Acute Sockets API, which is essentially identical to that of the low-level failure semantics of Task 1.1. These were:

1. An implementation of key parts of the classic *Chord* P2P algorithm.
2. An Acute library providing primitives for inter-site asynchronous channel-based communication and migration of running computations between sites, providing essentially the capabilities of the Nomadic Pict programming language but as a concise library.
3. An Acute library providing primitives based on the *Ambient* calculus.

All of these are decentralized implementations, with no central coordinator, and Acute provides guarantees of type safety even for interaction between distinct, separately compiled, and separately executed programs.

The latter two are contained in the Acute distribution, Deliverable 3.4, together with example programs that use them. The former was carried out at an earlier stage of the Acute development, so is not compatible with the final language, but provided essential feedback during development.

3.4 Models for anonymity analysis

At a still higher level of abstraction, we reported in D1.8 on an analysis of the properties of peer-to-peer systems for connection-based anonymity [36]. A journal paper on this work has now been accepted for publication [37].

3.5 Authority confinement by membranes

3.5.1 Problem context

The confinement problem is an old problem that any secure system should attempt to solve [19]. It states roughly that one can prove that some information will not be revealed by some part of a program, regardless of the rest of the program. For this goal to be possible we need some form of memory protection. At the OS level, this is done with the help special hardware, such as an MMU that ensures that the memory accessible by one process cannot be accessed by another process. At the language level also, this requires some memory safety. A language such as C where we can probe any address by casting integers to pointers is unable to solve the confinement problem. You have to read the whole code before you can state that the application does not leak some information. Memory safety is necessary but not the best we can do to help confinement. In most languages which are memory safe (even in those designed with strong security in mind like E [31]), you have to completely check the part of code which has access to some reference to the information, or when there are features meant to let you focus only on part of this code, they are so awkward that they are difficult to use to prove confinement (Java [43] is an example of such a language).

Confinement of information in the form of pure data is nearly impossible as it would require to remove all covert channels of the system. For this, we will only focus on authority confinement, as authority can be represented in the form of a reference to some black box which has no binary representation (or at least none available from within the language).

3.5.2 Related work

Confinement is quite common at the OS level. Apart from the memory isolation, there is also file confinement which is nearly always implemented with Access Control Lists (ACLs), or a restricted form of these, such as Unix permissions. This allows to prove that access to the file (but not to its contained information) is impossible from a process without the right permissions (invoked by the right user) or without the name of the file (but as the name is pure data it can be transmitted on a covert channel and nobody would say that a file is secure just because nobody knows its path).

At the language level, lexical scoping helps to prove confinement but is not enough as a reference can be given to an outer scoped variable, requiring more and more analysis. Another construct which is developed now, and sometimes called "membrane" or "glove" is purely based on capabilities. It consists of not giving a reference to the information to protect but to another object which stay in between. And every reference exchanged through it is encapsulated in the same way as the initial information was. This glove construct is more a wall. It is constructed element by element but lacks flexibility. There is no way to make it move. The only possible thing is to remove a brick, but this is enough to break the whole wall.

3.5.3 Formal calculi and language design work

This work is related to the work of securing the Oz language [39, 13] (the Oz-E project) and as such to capability systems in general such as E [24]. Some of the ideas are shared with ambients and mobile ambients (most notably the idea of an execution context, influencing the possibility to take certain actions or not). Those systems are in particular the M-calculus [35] and Kell-calculus [4]. Some earlier attempts at confinement were also interesting in comparison with our approach. Those includes factories (KeyKOS [16]) or rings (Multics [34]). Those were OS level primitives intended to makes confinement easier for the application programmer and as such are not as different to language primitive than pure OS level confinement is.

3.5.4 Membrane design guidelines

We introduce a concept of membrane in order to help solve the confinement problem. Our membrane concept is designed with the following goals in mind:

- ▷ **Simplicity.** A language is truly secure if it is easier to write things which are secure than it is to write things which are not. Security must be the default. An important precondition for this is that the model must be simple.
- ▷ **Fine granularity.** We really want to provide confinement at all levels starting from individual language references. This is also the granularity level of capability systems. So we will additionally seek a complete coherence with this security model and ensure that we do not break any of the properties which allow the capability models to be used successfully in the language.
- ▷ **Dynamicity.** Our solution is able to handle evolution in the trust relation between the components which are kept confined. Of course since trust is to be considered monotonic in security proofs (handling reducing trust in multi-agents systems isn't very different from handling mutable state in concurrent systems) we will only allow the confinement to be reduced, but in such a way as to keep a reduced confinement and not an all or nothing solution.
- ▷ **Strictness.** Our confinement is strict in the sense that once we establish that confinement as to be guaranteed for some information, all interaction with this information is prohibited unless explicitly allowed.

3.5.5 Overview of membrane concepts

A membrane provides an execution context. This means that every instruction is executed in the context of some membrane. Membranes constrain what the instruction can do with values that have token identity. The instruction is only allowed to interact with a token value if it has both a reference to that value (this is an obvious condition, as the language is memory-safe) and its membrane context contains the token value.

When a token value is created, it is automatically placed in the membrane of the creating instruction. In addition, there are three new language primitives directly involving membranes: membrane creation, exporting a token value outside of a membrane, and invoking execution inside a membrane.

- ▷ In addition to creating a new membrane, the membrane creation primitive returns two token values to the caller: the export capability and execution capability for the membrane. Both of these values are initially only available in the membrane of the creation primitive.

- ▷ The export capability is used in the export primitive. This primitive takes an export capability and a token value and if both are present in the membrane where the primitive is called, then the value is made also available in the membrane corresponding to the export capability.
- ▷ The execution capability is used in the execution primitive. This primitive takes an execution capability and a closure. If the execution capability is available in the membrane where the primitive is called (along with the instruction, if the form the instruction is in is considered as a token value) then the closure is executed in the context of the membrane corresponding to the execution capability.

These constructs fulfill the goals we established before in a simple and fine-grained manner.

3.5.6 Future plans

We are continuing this work in two directions: programmer support and distributed computing. For the first direction, we are investigating how to provide ways to make it easy for programmers to use membranes. Because of the fine granularity and the strictness of the confinement, using the membrane operations directly is often tedious as many things may need to be exported to some membrane before anything useful can be done in its context. This can be partly solved by preloading the membrane with a set of known to be generally useful and perfectly safe abstractions, but this doesn't scale very well and lack extensibility. Another solution could be to explicitly annotate (at creation time) some token values as carrying no authority (rather than marking those carrying authority as it wouldn't be strict anymore), and to let such values be freely usable. A third solution would be to be able to define groups or values which could be exported in one operation.

Another feature which would be very useful to program with membranes would be the ability to detect that some instruction in some membrane needs a value which is not available there, so that some form of security management could export it (for example, a PowerBox as defined in [40]), subject to certain security policies. We have started to work on this addition and the semantics of the operations we need is already clear.

The second direction for future work concerns distributed computing. We need to extend membranes to distributed contexts. Firstly, because it is the typical example of systems where confinement is actually weak and is in principle extremely necessary. Secondly, because security and distribution are presumably the future of computing. To extend membranes to distributed systems requires first some changes to their semantics to reflect the hierarchical nature of trust relations in such systems (one cannot trust a computation running on a non-trusted node). Possibly the space of membranes should no longer be flat as it is now but hierarchical. Supporting membranes on distributed systems also means developing cryptographic protocols to guarantee confinement on a network that can be tapped.

3.5.7 Conclusion

We are not doomed to live with insecure applications. In the same way (and mainly for the same reasons) that structured programming made correctness proofs much easier, well-structured trust relations between program parts will make security proofs easier. Research toward this has been going on for some time already, but now the software engineering and hardware advances are sufficient for solutions to be practical for everyday use. The membranes introduced in this work are a step toward this goal.

3.5.8 Acknowledgement

This research is supported by the following projects:

- ▷ PEPITO - PEer-to-Peer: Implementation and TheOry
- ▷ EVERGROW
- ▷ MILOS

4 Other work

Two position papers were published based on work in PEPITO:

- ▷ [20] James Leifer, Michael Norrish, Peter Sewell, and Keith Wansbrough. Acute and TCP: specifying and developing abstractions for global computation. In *Proceedings of the APPSEM II Workshop, Tallinn*. 2pp, April 2004. Available from <http://www.cl.cam.ac.uk/users/pes20/appsem-tallinn.ps>.
- ▷ [38] Peter Sewell and Keith Wansbrough. Applied semantics: Specifying and developing abstractions for distributed computation (grand challenge discussion paper – GC2, GC4, and GC6). Position paper for Grand Challenge meeting, Newcastle. 5pp, 2004. Available from <http://www.cl.cam.ac.uk/users/pes20/grandchallenge2004.pdf>.

5 Relations to other work packages

The work described here has been closely linked with that of Workpackages 2 and 3, and there have been strong inter-site links:

- ▷ The DKS verification of §3.1 (EPFL/KTH/SICS, with UCAM input in the early stages) obviously draws on the DKS algorithm itself, developed in WP2.
- ▷ The low-level failure semantics described here is carefully designed to fit together with the Acute development (WP3, INRIA/UCAM); the latter covers semantics above the Sockets API and the former that below. Together they support the reasoning experiments of §3.2 and the programming experiments of §3.3.
- ▷ The work on membrane confinement here is related to the work of securing the Oz language used in WP3.
- ▷ There is a large body of formal work on the foundations of distributed programming languages, supporting type-safe marshalling, dynamic rebinding, and dynamic update, that was initially placed in WP1 but reassigned during the project to WP3. This is largely an INRIA/UCAM collaboration.

6 Conclusion

At this point at the end of the project, drawing together work from WP1, WP2, and WP3, we have met our primary goals: we have shown that it is feasible to work with *fully specified systems*, with rigorous semantic foundations for the underlying TCP/UDP/Sockets networking, for the **Acute** programming language with marshalling support, and for the P2P algorithms written therein. We have also deepened the understanding of transactional mechanisms and P2P systems for anonymity.

This is a major advance on the previous state of the art, and opens up many directions for future research and development. Rigorous treatment of large bodies of executable code remains very challenging, but we hope that using our techniques, and with this proof-of-concept as an exemplar, the community will be able to develop better-understood, and hence more robust, P2P infrastructure.

References

- [1] ACM. *SIGCOMM 2001, San Diego, CA, 2001*.
- [2] M. Al-Metwally. *Design and Implementation of a Fault-Tolerant Transactional Object Store*. PhD thesis, Al-Azhar University, Cairo, Egypt, Dec. 2003. Available at <http://www.info.ucl.ac.be/people/PVR/Metwallythesis>.
- [3] M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. In L. Aceto and B. Victor, editors, *Proceedings of EXPRESS '00*, volume 39.1 of *ENTCS*. Elsevier Science Publishers, 2000.
- [4] P. Bidinger and J.-B. Stefani. The kell calculus: operational semantics and type system. In *Proceedings 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 03)*, Paris, France, November 2003.
- [5] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. Submitted for publication. Available at <http://www.cl.cam.ac.uk/users/pes20/Netsem/paper.pdf>, Feb. 2005. 15pp.
- [6] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Available at <http://www.cl.cam.ac.uk/users/pes20/Netsem/tr.pdf>, Feb. 2005. vi+96pp.
- [7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The specification. Available at <http://www.cl.cam.ac.uk/users/pes20/Netsem/alldoc.pdf>, Feb. 2005. xxv+380pp.
- [8] A. P. Black, V. Cremet, R. Guerraoui, and M. Odersky. An equational theory for transactions. In *Proceedings of FSTTCS03: 23rd Conference on Foundations of Software Technology and Theoretical Computer Science, Mumbai (Bombay), India*, December 2003. Full version available as EPFL technical report IC/2003/26, 2003. <http://lamp.epfl.ch/~cremet/publications/fsttcs03.ps>.

- [9] J. Borgström, U. Nestmann, L. Onana Alima, and D. Gurov. Verifying a structured peer-to-peer overlay network: The static case. Technical Report IC/2004/76, EPFL, Sept. 2004. Available from http://ic2.epfl.ch/publications/documents/IC_TECH_REPORT_200476.pdf.
- [10] J. Borgström, U. Nestmann, L. Onana Alima, and D. Gurov. Verifying a structured peer-to-peer overlay network: The static case. In C. Priami and P. Quaglia, editors, *Proceedings of Global Computing 2004*, volume 3267 of *Lecture Notes in Computer Science*, pages 251–266. Springer-Verlag, 2005. Available from <http://lampwww.epfl.ch/~uwe/doc/borgstroem.etal:gc-2004.pdf>.
- [11] B. Carton and V. Mesaros. P2PS: Peer-to-Peer System Library, Oct. 2004. Available at <http://www.mozart-oz.org/mogul/info/cetic.ucl/p2ps.html>.
- [12] M. Compton. Stenning’s protocol implemented in UDP and verified in Isabelle. In *Proceedings of The Australasian Theory Symposium*, 2005. Available at <http://www.cl.cam.ac.uk/users/pes20/Netsem/stenning.pdf>.
- [13] T. M. Consortium. Mozart/Oz website. <http://www.mozart-oz.org>.
- [14] W. Fokkink, J. F. Groote, and M. Reniers. Process algebra needs proof methodology. *EATCS Bulletin*, 82:109–125, Feb. 2004.
- [15] W. J. Fokkink, J. F. Groote, J. Pang, B. Badban, and J. van de Pol. Verifying a sliding window protocol. In C. Rattray and S. Maharaj, editors, *Proceedings of AMAST’2004*, volume 3116 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2004. Full version available as Technical Report SEN-R0308, CWI Amsterdam, 2003.
- [16] B. Frantz. KeyKOS - A Secure, High-Performance Environment for S/370. In *Proceedings of SHARE 70 I (SHARE Inc, Chicago)*, pages 465–471, February 1988.
- [17] K. Glynn. Extending the oz language for peer-to-peer computing. PEPITO project deliverable D3.7, IST-2001-33234, Dec. 2004. Available at <http://www.sics.se/pepito/deliverables.html>.
- [18] K. Glynn. P2PKit Library, Oct. 2004. Available at <http://renoir.info.ucl.ac.be/twiki/bin/view/INGI/P2PKit>.
- [19] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.
- [20] J. Leifer, M. Norrish, P. Sewell, and K. Wansbrough. Acute and TCP: specifying and developing abstractions for global computation. In *Proceedings of the APPSEM II Workshop, Tallinn. 2pp*, Apr. 2004. Available from <http://www.cl.cam.ac.uk/users/pes20/appsem-tallinn.ps>.
- [21] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM 373, MIT Press, Nov. 1998.
- [22] V. Mesaros, B. Carton, and P. Van Roy. P2PS: Peer-to-peer development platform for Mozart. In *Proc. of the 2nd International Mozart/Oz Conference – MOZ 2004*, Oct. 2004.
- [23] V. Mesaros, R. Collet, K. Glynn, and P. Van Roy. A transactional system for structured overlay networks. Research Report RR2005-01, Université catholique de Louvain, 2005. Available at <ftp://ftp.info.ucl.ac.be/pub/reports/2005/rr2005-01.pdf>.

- [24] M. Miller, M. Stiegler, T. Close, B. Frantz, K.-P. Yee, C. Morningstar, J. Shapiro, N. Hardy, E. D. Tribble, D. Barnes, D. Bornstien, B. Wilcox-O’Hearn, T. Stanley, K. Reid, and D. Bacon. E: Open source distributed capabilities, 2001. Available at <http://www.erights.org>.
- [25] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, 2002.
- [26] U. Nestmann and R. Fuzzati. Unreliable failure detectors via operational semantics. In V. A. Saraswat, editor, *Proceedings of ASIAN 2003, LNCS 2896*, pages 54–71, Dec. 2003. <http://lampwww.epfl.ch/~uwe/doc/nestmann.fuzzati-asian03.pdf>.
- [27] U. Nestmann, R. Fuzzati, and M. Merro. Modeling consensus in a process calculus. In R. Amadio and D. Lugiez, editors, *Proceedings of CONCUR 2003*, volume 2761 of *LNCS*. Springer, Aug. 2003.
- [28] M. Norrish, P. Sewell, and K. Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proceedings of the 10th ACM SIGOPS European Workshop (Saint-Emilion)*, pages 49–53, Sept. 2002.
- [29] L. Onana Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS (N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for P2P applications. In *CCGRID 2003*, pages 344–350, 2003.
- [30] L. Onana Alima, A. Ghodsi, S. El-Ansary, P. Brand, and S. Haridi. Design principles for structured overlay networks. Technical Report ISRN KTH/IMIT/LECS/R-03/01–SE, KTH, 2003.
- [31] T. O. E. Project. E-Rights.org, home of E. <http://www.e-rights.org>.
- [32] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM 2001, San Diego, CA* [1].
- [33] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.
- [34] J. H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, 1974.
- [35] A. Schmitt and J.-B. Stefani. The m-calculus: A higher-order distributed process calculus. In *In Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages (POPL’03)*, New Orleans, LA, USA, January 15-17 2003.
- [36] A. Serjantov and P. Sewell. Passive attack analysis for connection-based anonymity systems. In *Proceedings of ESORICS 2003: European Symposium on Research in Computer Security (Gjøvik)*, LNCS 2808, pages 116–131, Oct. 2003. http://www.cl.cam.ac.uk/users/aas23/papers_aas/conn_sys.ps.
- [37] A. Serjantov and P. Sewell. Passive attack analysis for connection-based anonymity systems. *International Journal of Information Security (IJIS)*, 2005. To appear.
- [38] P. Sewell and K. Wansbrough. Applied semantics: Specifying and developing abstractions for distributed computation (grand challenge discussion paper – GC2, GC4, and GC6). Position paper for Grand Challenge meeting, Newcastle. 5pp, 2004. Available from <http://www.cl.cam.ac.uk/users/pes20/grandchallenge2004.pdf>.

- [39] F. Spiessens and P. Van Roy. The Oz-E project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004*, volume 3389 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.
- [40] M. Stiegler and M. S. Miller. A capability based client: The darpabrowser. Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex, Inc., June 2002. Available at <http://www.combex.com/papers/darpa-report/index.html>.
- [41] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Wide-area cooperative storage with CFS. In *Proc. of the ACM Symposium on Operating Systems Principles*, October 2001.
- [42] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM 2001, San Diego, CA* [1].
- [43] Sun Microsystems, Inc. Security and the Java Platform. <http://java.sun.com/security>.
- [44] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Mar. 2004.
- [45] K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proceedings of ESOP 2002: the 11th European Symposium on Programming (Grenoble)*, LNCS 2305, pages 278–294, Apr. 2002.