



**PEPITO**  
**IST-2001-33234**  
**PEer-to-Peer Implementation and TheOry**

**Deliverable no: D1.1**

## **Required foundations for peer-to-peer systems**

REPORT VERSION: first

REPORT PREPARATION DATE: 2003.06

CLASSIFICATION: Public

DELIVERABLE NO: D1.1      DUE DATE: Month 18      DELIVERY DATE: Month 18

PROJECT START DATE: 2002.01.01      PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: UCAM

PARTICIPATING PARTNERS: EPFL, INRIA, UCAM

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  
'Information Society Technologies' Programme (1998-  
2002)**

Project Number: IST-2001-33234  
Project Acronym: PEPITO  
Title: PEer-to-Peer Implementation and TheOry  
Deliverable No: D1.1  
Required foundations for peer-to-peer systems  
Due date: project month 18  
Delivery date: 2003-06-30

Responsible Partner: UCAM  
Participating Partners: EPFL, INRIA, UCAM

26th June 2003

Prepared by Peter Sewell, with input from James Leifer, Uwe Nestmann, Andrei Serjantov, and Keith Wansbrough.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>P2P systems – the characteristics</b>	<b>3</b>
<b>3</b>	<b>Applicable Foundations</b>	<b>4</b>
<b>4</b>	<b>Key issues</b>	<b>5</b>
4.1	Specification of Subtle Distributed Algorithms . . . . .	5
4.2	Verification . . . . .	6
4.3	Fault Tolerance . . . . .	8
4.4	Mobility . . . . .	9
4.5	Programming Language Design . . . . .	10
<b>5</b>	<b>Example: Chord Modelling</b>	<b>13</b>
<b>6</b>	<b>Example: Analysis of Anonymity Systems</b>	<b>15</b>
<b>7</b>	<b>Relations with Other Work Packages</b>	<b>16</b>
<b>8</b>	<b>Interim Progress</b>	<b>17</b>
<b>9</b>	<b>Conclusion</b>	<b>18</b>

## 1 Introduction

The original PEPITO Technical Annex proposed, in a task of WP1, to

...identify key issues of peer-to-peer computation and collaboration by distilling them in a *formal calculus*.

There were accompanying deliverables D1.1 (due month 18) “Requirements for a peer-to-peer calculus” and D1.2 “A formal calculus for peer-to-peer computation and collaboration” (due month 36). As we said in our letter of April 11, 2003, in the light of the first year we now realize that this goal – of developing a single all-encompassing P2P calculus – is not scientifically well-motivated. The number of different issues to be addressed mean that any such calculus would be too large and complex to study mathematically, but neither would it provide a useful tool for implementors. Instead, it is much more fruitful to (a) identify the key issues of P2P systems, as originally planned, and then (b) to focus clearly on foundational work to address them. The latter requires developing a *variety* of formal calculi, languages, and reasoning techniques. In this D1.1 report (retitled “Required foundations for peer-to-peer systems”) we pursue (a), discussing the issues, how they arise from the characteristics of P2P systems, the state of the art in addressing them, and what needs to be done. Many, but not all, are being worked upon in PEPITO. We expect the report D1.2 (retitled “Survey of formal challenges and solutions for peer-to-peer computation”) will put this and other work into context.

### Outline

This report places the foundational work of PEPITO, the initial work of which was described in Deliverable D1.7 “First Progress Report on Formal Models”, in the broad perspective of P2P systems. We begin by recapitulating the characteristics of peer-to-peer systems. We then discuss the nature of foundational work in general, highlighting the different kinds of contribution it can make. The main body of the report, Section 3, discusses the main problem areas in which foundational work is required for systems with the P2P characteristics: techniques for Specification and Correctness of the subtle distributed algorithms that arise in these systems; Probabilistic and Statistical properties thereof; Fault Tolerance; Mobility; and Programming Language Design. We continue by sketching a case study, outlining the mathematical structure that would be required for rigorous specification and verification of the correctness of the Chord Distributed Hash Table algorithm. We then discuss another challenging application area, that of stating and proving properties of P2P systems for anonymity – an intrinsically statistical property, related to traffic analysis. We conclude with a discussion of the relationships to the other Work Packages of PEPITO and a brief summary.

## 2 P2P systems – the characteristics

The notion of “peer-to-peer system” is not subject to precise mathematical definition. Instead, peer-to-peer is a class of distributed systems with some common architectural and organisational aspects. The PEPITO deliverable D2.7 “First progress report on distributed algorithms” surveys a number of existing systems, widely accepted to be P2P, and proposes a definition of P2P systems as those that share the following properties:

- ▷ Decentralization: no single centralized coordination nor administration;
- ▷ Scalability: the system size can grow arbitrarily while keeping the system performance acceptable.

- ▷ Adaptation: the system adapts to frequent changes in content and load;
- ▷ Self-organization: despite frequent arrival and voluntary departure of nodes, (nearly) optimal configurations emerge without manual intervention;
- ▷ Fault-tolerance: the impact of node/communication failures on the overall system is minimized;
- ▷ Equal functionality: each node can play the role of client, server, router and cache;
- ▷ Mobility: nodes do not have fixed IP addresses.

We take a broad view of P2P, encompassing not just the traditional content-sharing systems, distributed-hash-table indexing and the like, but also other decentralized systems at various scales.

In addition, it is important to note that the most fundamental characteristic of P2P systems is that they are *distributed* systems, indeed, that they are exemplary *Global Computing* systems. As we shall see, many of the most important problems for P2P systems arise from this. Solutions to them will therefore contribute to development of robust P2P systems but are not narrowly-specific to P2P; they will also be widely applicable to other Global Computing scenarios. As a methodological point, however, the general problems must be addressed in the context of specific examples; the P2P domain provides a rich and developing source of these examples.

### 3 Applicable Foundations

Theoretical and practical work on distributed systems both have a long history but, regrettably, the two have not often been developed hand-in-hand. There are three main ways in which foundational work can be applied:

- ▷ Specification: expressing precisely the behaviour of APIs and key algorithms;
- ▷ Verification: proving correctness and performance properties (temporal and/or probabilistic);
- ▷ Design: providing a basis for language and API design.

Expanding on these:

- ▷ Specification: a formal specification provides valuable documentation, reducing the barrier to entry and thus enabling the use of the API by a wider community than just the designers. This is of more than merely educational benefit; it aids the rapid rollout of new technology, and so increases the likelihood of market dominance. By providing an explicit contract between implementor and programmer, it also permits heterogeneity of implementation, historically a critical factor in determining which technologies succeed and which fail.
- ▷ Verification: such foundations allow the possibility of *proving* that a system has the desired qualitative properties (correctness) and quantitative properties (performance).

Correctness guarantees that a system will work, and be stable and dependable. This can include self-organisation or fault-tolerance properties. Correctness has been the focus of much work on “formal methods”, but, we emphasize here, is only one aspect of foundational work.

The verification of performance properties is similar to the verification of correctness properties, but even more challenging. Foundational work in this domain addresses the techniques and

possibility of proving that a system has the desired temporal and/or probabilistic performance properties, guaranteeing that it will be usable and scalable.

Most interesting, and also most challenging, are combined properties where correctness is “only” statistically achieved to some quantifiable degree under probabilistic assumptions.

- ▷ Design: foundational work on distributed systems in general and peer-to-peer in particular provides the tools required for the design of new languages and APIs. New features may be prototyped and developed in isolation, before integration into a larger system. Critical functionality and hidden symmetries that should be explicit in the interface (the ‘right way to do it’) can be discovered early in the development cycle, before the cost of change is too great. Furthermore, standard techniques such as approximate static analysis (often in the form of type systems) may be used to straightforwardly rule out large classes of common bugs, making program development more efficient.

Foundational work can take various forms: small *calculi*, typically with operational semantics, can be used to explore the design of particular constructs; larger programming and/or specification *languages* integrate a collection of ideas; *semantic techniques* and *reasoning methods* provide ingredients for specification and proof, *approximate static analysis* and *type systems* are used to express behavioural invariants of systems.

We end this section with an apt quotation from *The Art of Unix Programming*.

Formalization often clarifies a task spectacularly. It is not enough for a programmer to recognize that bits of his task fall within standard computer-science categories – a little depth-first search here and a quicksort there. The best results occur when the nub of the task can be formalized, and a clear model of the job at hand can be constructed. It is not necessary that ultimate users comprehend the model. The very existence of a unifying core will provide a comfortable feel, unencumbered with the why-in-hell-did-they-do-that moments that are so prevalent in using Swiss-army-knife programs.

– Doug McIlroy (in [Ray03, §4.2]).

## 4 Key issues

There are many problems in the design and implementation of P2P systems which call out for foundational work. In this section we give a high-level overview of those we think most important, and of how they should be addressed. Most (but not all) are the subject of work in PEPITO.

### 4.1 Specification of Subtle Distributed Algorithms

P2P systems are characterised by subtle distributed algorithms. Decentralized administration, and the need for fault-tolerance in large systems, lead to non-hierarchical algorithms. Adaptation and self-organisation introduce further complexity. In addition, all the classic difficulties arising from distribution are still present – concurrency, nondeterminism, obscure low-level communication APIs, large and hard-to-understand state spaces, etc.

Unsurprisingly, this concentration of issues makes design and implementation difficult. Moreover, due to the subtleties of P2P algorithms, it is even a matter of debate to choose and work at appropriate levels of abstraction and formality, yet the present state of the art does not ease the task.

Specifically, current approaches to distributed algorithm design abstract away from details in too many places, leaving a wide and treacherous gap to be traversed by the implementor: (1) They abstract from the low-level protocols, substituting idealisations such as synchronous messaging or perfectly reliable streams. (2) They abstract from the high-level API provided by the service, substituting model states and informal initialisation conditions. (3) They abstract from the real code of the implementation, substituting ambiguous pseudocode. (4) They abstract from formal proof, substituting informal prose arguments. To achieve reliable infrastructure, we must address as many of these points as possible.

To do so, it is necessary to push the boundaries of specification, of validation, and of formal semantics. Of course, a great deal of work has been done in these areas – far too much to review here – but current techniques are still insufficient, tools underdeveloped, and experience lacking. Particularly seriously, in our view, there is a lack of semantic integration: we have

- ▷ algorithms commonly expressed in pseudocode (often ambiguously);
- ▷ algorithms sometimes verified (and expressed) in automata-theoretic terms; and
- ▷ algorithms implemented in large bodies of C, C++ or Java.

The relationships between these are not made precise. Instead, we need good idioms, languages and proof techniques that can bring the three together. This would enable properties of actual implementations, running over real UDP/TCP and sockets, to be rigorously stated, validated, and (in some cases) proven; this would significantly increase the trustability and reliability of infrastructure for distributed computation.

## 4.2 Verification

Many of the pragmatic benefits of foundational work do not require full formal verification of P2P algorithms, but arise already from formally precise specification of their desired properties and of the underlying protocols. Fully formal verification is demonstrably feasible, but its cost means that it is currently only worthwhile for carefully-chosen examples – not so large as to be infeasible, nor too small as to be trivial. To develop adequate formal foundations for complex P2P systems, with all their subtleties, it is useful to start from available techniques and push them gradually towards P2P systems.

### Correctness Properties

Our prior and ongoing work has addressed several carefully-chosen examples: rigorous formalisation of consensus and failure detector algorithms, simple failure detection above the real UDP sockets interface, and algorithms for location-independent communication between mobile entities, in the *Nomadic Pict* setting (a number of the latter algorithms were peer-to-peer, though before that term became ubiquitous).

For example, in the light of the above-mentioned lack of formality in state-of-the-art techniques, our exercise on the formalisation of Consensus in the presence of failure detectors tried to bridge the gap between two fields and communities: the one of Distributed Algorithms on the one hand, traditionally using pseudocode for the specification of algorithms and their underlying abstractions, and prose arguments in some ad-hoc logics for the proofs of correctness properties; and the field of Concurrency Theory on the other hand (notably the field of process calculi, which is widely populated not only within the PEPITO project, but in most projects of the whole Global Computing initiative),

where algorithms are described as mathematical entities with formally precise semantics, and proofs that are carried out within the chosen formal setting.

Consensus was a good example in the above-mentioned sense: not too big, not too small, sharing a number of aspects with algorithms as used in P2P infrastructures, but simpler.<sup>1</sup> Our work only addresses a fixed number of fault-tolerant processes in a reasonably closed system, and the required properties were only related to correctness, not to performance (see below). But already in this exercise, the required foundational work led us to establish a method and setting to address formal verification of non-trivial distributed algorithms. The essence was to find a way to connect the formal process calculus description (substituting the pseudocode of traditional approaches) to a formally-defined global-view data structure (substituting the informal prose argument of traditional approaches). Then, carrying out a formal proof became feasible and comprehensible.

We can now try to push our techniques for the verification of correctness properties towards algorithms underlying P2P systems. The P2P field provides a rich source of examples from which to select. One possibility is to look at distributed hash table (DHT) algorithms such as that of Chord or (slightly more complex) the PEPITO-developed DKS. Later in this document we consider the former in some detail, building on the essential experience of our Consensus exercise (for formalization idioms and proof techniques) and our work on UDP/TCP (for the semantics of real-world failure modes).

### Performance Properties

A simple performance measure that is often employed in distributed algorithms, and in particular P2P algorithms, counts the number of messages needed to achieve a given goal. For example, the number of messages that are needed in order to look up an entry in a DHT, or the number of messages that are needed in order to maintain a DHT in the presence of joins and leaves of nodes. For example, a typical advantage of the DKS developed in PEPITO over its predecessors like Chord is the reduced number of maintenance messages by the *correction-on-use* technique.

In static closed systems, it is usually possible to provide (upper and lower) bounds. In open dynamic systems, we rather have to refer to probabilistic and statistical properties, because the assumptions on which the bound guarantees are computed become moving targets. In fact, it is frequently the case that providing absolute guarantees of desired properties is prohibitively difficult, expensive, or impossible, and in fact unnecessary. ‘Near enough is good enough’ – at least, when ‘near enough’ is precisely quantifiable.

The two-generals problem [Jim Gray, Notes on Database Operating Systems, in LNCS 60, 1978, §5.8.3.3.1] demonstrates that fault detection, consistency, and commitment are all impossible to achieve perfectly in a distributed environment; however, in practice there are algorithms that achieve them with any desired probability of accuracy. Traditional reasoning techniques do not suffice to reason about such protocols; instead, statistical properties of system behaviour must be taken fully into account.

The canonical example is hashes, as in our DKS, or DHTs in general. In a DHT document-sharing system, we use the hash of a document to determine where it should be stored; we use hashes of node identifiers to determine the ring topology. We rely on probabilistic properties of the hash function to achieve a near-uniform distribution of documents over nodes, without hot-spots, and to achieve a balanced topology. To provide a non-probabilistic *guarantee* of fair distribution would require far more communication, and likely be much less resilient to arrival/departure and failure.

---

<sup>1</sup>Consensus algorithms are not required for lower levels of P2P infrastructures, but any group-oriented application running on top of some P2P infrastructure profits from the availability of this distributed programming abstraction.

Another example, again taken from DHTs, concerns the impact of node joins on lookup performance. In Chord, “unless a tremendous number of nodes joins the system, the (problematic) number of nodes between two old nodes is likely to be very small, so the impact is negligible” [SMLN<sup>+</sup>03]. This is then taken to sketch the proof of a theorem, literally, “*with high probability*”, a term that appears very often in the cited paper. Clearly, there is a strong need for foundational work to be carried out on formalisms that are enhanced with probabilistic capabilities, to precisely model the above prose assumptions and conclusions, and to connect these formalisms unambiguously to the programming languages (or process calculi) that are used for the precise specification of the algorithms’ code.

A number of probabilistic and stochastic process calculi have been developed over the past 15 years (*e.g.* the PEPA of Hilston et al, and the IMC of Hermanns), and automata-theoretic models have also been considered. Probabilistic algorithms such as Rabin-Miller primality testing algorithm have been verified in automated proof assistants (in work by Hurd); this work demonstrates how much mathematical underpinning is required, as it developed a usable theory within the HOL proof assistant.

Reasoning about systems for private/anonymous communication, often structured in a P2P fashion to avoid *administrative* single points of failure, is necessarily statistical; we describe some of that taking place in PEPITO later in this document.

### 4.3 Fault Tolerance

P2P systems must tolerate many different kinds of failure while maintaining critical system properties such as stability and consistency. The kinds of failure that must be tolerated include:

- ▷ Host failure: peers are ordinary machines in the network, and so we cannot assume that measures appropriate for servers, such as reliable operating systems, uninterruptible power supplies and 24-hour support staff, are in place. On a more basic level, users switch their machines off, both deliberately and accidentally. As D2.7 shows, hosts may also fail due to political and/or legal action.
- ▷ Process failure: applications may be shut down or killed by users at any time; they may run out of local resources, or crash as a result of bugs. While security is largely outside the scope of this project, peers lack central administration, and so are under the control of potentially malicious users. Such attackers may alter the behaviour of a process arbitrarily.
- ▷ Link failure: P2P systems build up a complex overlay network of connections between peers, with much more exposure to unreliable edge links than a client/server star topology. In particular, failure of a particular edge link affects not just the node it connects to the network, but all the other nodes that are connected to that node through the link. This form of link failure is closely related to host failure. Edge links fail not just because of their lower importance from the point of view of network operators, but because of their inherent transience – dynamic assignment of addresses, power failure, and intentional disconnection for mobility or disconnected operation. Link failure also arises from intrinsic properties of the Internet: packets may be lost or arbitrarily delayed, and in fact packet loss is fundamental to the operation of the network, signalling congestion and triggering rate limiting mechanisms.

Amongst other things, these points amplify the importance of decentralization, adaptation, self-organisation, and equal functionality. Nothing must be lost as a result of a failure, and the overlay structure must be maintained.

In order to build a formal treatment of fault tolerance, we must begin by understanding and specifying the precise behaviour of these failures. What kinds of failures can occur? How do they present themselves? What error codes are returned by the network and by the API, if any? What guarantees are provided by the lower layers? Do hosts, processes, and links fail permanently or temporarily? What bounds can be placed on failure rates and durations? Are failures uniform or clustered? What properties of an operational link can be relied upon, in order to detect failure as deviation? Need we consider only fail-stop, or must we additionally consider byzantine failure, or attack?

With this understanding, we may proceed to develop algorithms that tolerate such behaviour, through detection, through replication or redundancy, through transactions, through consistency protocols, through cryptographic techniques, and so on. Ideally, these algorithms will hide the failure completely from the layers above; in other cases, they will greatly reduce the set of possible failure modes and improve feedback.

For example, one may guard against network partition (and allow disconnected operation) by replicating state across the network, and using some consistency protocol to resolve replicas when the network is reconnected (*e.g.*, Unison). One may preserve an overlay topology by detecting failure with a heartbeat protocol, and storing enough state in neighbouring nodes that they may unilaterally perform the leave operation themselves for the failed or disconnected node. One may present a much simpler model of distributed operation, hiding its intrinsic partial, distributed, byzantine failure modes, by encapsulating a transactional store and the two-phase commit protocol, providing a simple interface with two failure modes: the whole distributed transaction atomically either commits (succeeds completely) or aborts (fails completely).

Having used our foundational specification tools both for our problem domain and for the proposed solution, we may proceed to analyse correctness and performance. The properties we prove about the algorithms will often be statistical, not absolute, as we noted above, *e.g.*, ‘connectivity will be maintained despite failure of up to  $k$  nodes’, ‘consensus will be reached within 5 rounds with probability 99%’.

Finally, the design tools provide a clear path to implementation of the algorithms developed, and integration of them into existing programming environments.

#### 4.4 Mobility

Mobility is central to many P2P applications. Users work from multiple locations, hosts (especially laptops, PDAs, and embedded computers) move within and between networks, and network addresses are dynamically assigned. In all these cases, connectivity (and locatability) must be maintained, and both system and host must adjust to the new situation. In cases such as ad-hoc conferencing, mobility is the reason for the application’s existence, and it is within a specific mobile context that a P2P overlay network must be constructed and used. These requirements greatly complicate the distributed algorithms that must be used and the failure modes that occur, driving research in those areas.

A new range of applications are enabled if we allow finer-grained mobility, of *code*, either small fragments or entire running processes (consider Java applets as a simple, non-P2P, client/server example). The use of this technology in P2P opens up many intriguing possibilities. Users may delegate authority to *agents*, programs which move through the network towards the resources they require, perform the desired operations there where they are not subject to network latency, and then return with the results. Load balancing may be performed by cloning hotspot nodes (such as index servers, or nodes hosting a Slashdotted document) and migrating them through the network. Queries or other operation parameters may be specified flexibly and generally by code fragments. Processes may migrate away from hosts that are about to be shut down or disconnected, or may checkpoint themselves

to achieve this even in the event of failure.

The ability to package up fragments of code, transport them to another location, and continue executing them, and have the results make sense, relies on both careful programming language design (and algorithm correctness), and careful engineering. There are many issues, ranging from technical to social:

- ▷ Rebinding: how should a program fragment, which inevitably refers to local resources (memory cells, application-specific and system libraries, devices and so on) be interpreted at a new location? When should these references be rebound to local analogues, and when should the referenced resources be proxied, duplicated, or transported along with the fragment?
- ▷ Identity: if data is to be exchanged between two separated instances, when should they be considered ‘the same program’, and communication between them considered safe?
- ▷ Efficiency: how can the system avoid transferring code (especially required support libraries) more often than necessary, while maintaining consistency?
- ▷ Security: is it safe for a user to run code from elsewhere in the system on their machine? How can they be sure of its provenance and integrity?
- ▷ Cost: who pays for the resources (time, memory, bandwidth) used by mobile code?

Many of these issues motivate the programming language design work we consider below.

## 4.5 Programming Language Design

Two of the key characteristics of P2P systems are the *distributed* nature of the nodes and the *decentralized* control of their construction and deployment. Both characteristics are a challenge to accommodate with existing programming languages and therefore force us to consider new language designs.

A single computer in isolation is no longer useful. Yet we still, 35 years after the start of networking, program for single computers. Conventional programming languages (C, C++, Java, Fortran) do not deal with the issues of dynamic (re)configuration and adaptation, versioning, binding and composition, or code and thread mobility. Nor do they support the rich data types one might wish for non-numerical computation. Worse, many design and coding errors are not discovered until run-time – and discovering errors as early as possible is especially important in distributed systems, as they are extremely hard to debug (as a consequence of their nondeterminism and the lack of good tools), and patching all sites may be costly. What is needed now is a good understanding of the issues listed above, leveraging experience on static (compile-time) error detection to produce new language designs for flexible type-safe distributed computation.

When breaking a monolithic program into distributed pieces, the easy local manipulation of data and control is replaced by network communication. Moreover, this communication may be between units that do not have simple trust relationships. These break many of the principal implicit assumptions of non-distributed programming:

### **Implicit assumption: “Structured data can be passed easily and safely”**

Suppose, for example, one transforms an editor into a distributed collaborative editing system or a centralized dungeon game into a distributed multi-player game. In the non-distributed version of

these, it would be natural to use (fancy) abstract data types for the components. In essentially all programming languages it is easy to pass structured data by a pointer; moreover for languages with strong typing, such as Java, the safe part of C#, and ML, it is safe to do so. The latter languages guarantee both basic safety, namely that the all code that dereferences the pointer has a consistent view of the way the data is laid out in memory, and, *abstraction safety*, namely code only manipulates the data through the published interface of the data type, thus preserving the programmer's invariants.

For distributed programming, by contrast, one is faced with choices as to how to communicate data between machines. One could fix on a character wire format, or use an existing structured format (RPC, CORBA, XML/SOAP). This strategy suffers from several shortcomings: one needs to write printing functions for sending data and parsers for receiving it, an encoding/decoding that is not necessarily natural; it may be difficult to specify exactly the data structure invariant in the syntax of the format and therefore one needs to write a verifier for checking that each received input conforms to the specification; finally, whenever the data structures change during the development of the system, the printing functions, parsers, and verifiers need to be updated correspondingly.

Alternatively, one could marshal and unmarshal the values in a language-dependent way but without any type check or only internal type representation checking. In the former case, no safety is guaranteed at all. In the latter case, it is impossible to detect insidious bugs that break abstraction safety, for example receiving a search tree whose key ordering is opposite to the expected one. One might try to fix these deficiencies by including the source code's type names in the marshalled package and do a name check at unmarshal time. This works well if there is a *single* source code that is run on multiple machines but not if the communicating programs differ. Thus, we need a more subtle way to check that the sender and receiver have compatible invariants by comparing the parts of the programs that implement the abstract data type.

In [LPSW03] we develop compile-time and run-time semantics for marshalling that guarantee abstraction-safety between separately-built programs. This obtains a namespace for abstract types that is global, i.e. meaningful between separate programs, by *hashing* module definitions. These hashes are essentially fingerprints of the code that comprises the module, are included as a package with the marshalled value when sending a message, and are compared by the receiver to the expected type in a dynamic check.

With language support for safe marshalling, one can implement a variety of useful mechanisms above the standard (byte-string) primitives for network communication and persistence. For example: (1) In the existing distributed languages Jocaml [JoC] and Nomadic Pict [SWP99] a single program can dynamically distribute computations, which can then interact via typed channels, but unsafe "name servers" are required to bootstrap connections *between* programs. Type- and abstraction-safe marshalling would enable such name servers to be expressed in a safe way. (2) More generally, safe marshalling would enable one to code up a variety of communication abstractions, such as typed channels with differing behaviour (asynchronous, unicast, multicast, ...), within a high-level language; they would then be automatically guaranteed to be safe.

**Implicit assumption: "There is a single definition/version for all abstract types during a run"**

The previous discussion argued that communication between distributed programs requires language support for ensuring abstraction safety. Sometimes, however, we need to either force new versions to be compatible with old or, conversely, incompatible in ways that the language could not infer automatically.

Systems that are large-scale and have no centralized administration cannot synchronise software updates; hence we must ensure communication deals properly with changing versions. A simple

case is that in which the representation of the abstract type is unchanged and where the programmer asserts that the two versions have compatible invariants, so it is legitimate to exchange values in both directions. This may be the case even if the two are not identical, *e.g.* for an efficiency improvement or bug fix. Here we require language support for forcing the old and new types to be identical.

Dually, sometimes it is desirable to *force* a type change between builds even when the code remains identical, to prevent confusion between old and new communicated values. For example, one may have several distributed deployments of the same application which should be kept logically isolated.

### **Implicit assumption: “Data pointers come from a unique memory store”**

When passing a structured data within a single program, the pointers contained within are meaningful for the receiver since it shares a unique memory store with the caller. By contrast, when sending a message containing structured data across the network, this assumption is no longer valid. There are many possible choices and therefore rather than fixing on a particular choice, one wants language support for all of them:

- ▷ The pointer’s target is copied with the message, so it is available at the receiver side.
- ▷ The pointer is rewritten during the communication (a kind of pointer swizzling) to point to a target already existing in the receiver’s store.
- ▷ The pointer is rewritten by the communication to a stub on the receiver’s side that maintains a remote connection to the original target in the sender’s store.
- ▷ The communication is considered an error since there is a “dangling” pointer.

In all cases, one wants to be able to choose the treatment per pointer, perhaps by statically defined regions on the source code. One would then be able to copy all targets for pointers that are declared within one region, rewrite pointers outside this region but inside another, and get an error for any pointers from further afield.

### **Implicit assumption: “Code pointers come from a unique memory pool”**

Rebinding of *code pointers* may also be required when values or computations are marshalled from a running system and moved elsewhere, either by network communication or via a persistent store. These may be both ‘external’ identifiers of system-calls or language run-time library functions, and, more interestingly, ‘internal’ identifiers from application libraries which exist in the new context. Such libraries should not be automatically copied with values that use them, both for performance reasons and as they may have location-dependent behaviour (*e.g.* overlay routing functions). Moreover, a value may be moved repeatedly, and the set of identifiers to be rebound may change as it moves. For example, it may be desirable to acquire an organisation-specific library that, once resolved, should be fixed and carried with code moved within that organisation.

Flexible control of dynamic rebinding can also support *secure encapsulation* of untrusted code, by allowing access only to sandboxed resources. For example, when loading an untrusted applet, we may bind its `open` identifier to a `safe_open` function that only opens files in the `/tmp` directory. On the other hand, we want the flexibility to link trusted code with the unconstrained `open` function.

**Implicit assumption: “Code does not change at run-time”**

Systems that must provide uninterrupted service must be *dynamically updated* to fix bugs and add new functionality – essentially by loading new code into the program and then dynamically rebinding some of the existing identifiers to the new definitions, as supported by the Erlang language.

Our work [BHS<sup>+</sup>03, BHSS03] addresses both dynamic rebinding and dynamic update in a uniform semantic setting.

**Designing distribution-enabled P2P languages**

These various novel features must be first developed in isolation, with proofs of soundness theorems, and then integrated into full language designs. Such languages will eliminate much syntactic and semantic ‘noise’ from distributed programming, contributing to the rapid construction of robust systems.

## 5 Example: Chord Modelling

As an example, we now sketch the mathematical structure that would be required for verification of an executable description of the Chord Distributed Hash Table algorithm [SMK<sup>+</sup>01].

The first step is to express the algorithm precisely, in an executable language for which we can give an operational semantics. The original paper [SMK<sup>+</sup>01] uses informal pseudocode, and the implementation involves a large body of C++, for which an operational semantics is infeasible. Instead, one can express the per-node code in a monomorphic fragment of OCaml (MiniCaml), for which typing and semantics is straightforward. Communication, expressed in the original paper with an informal RPC mechanism, could either be coded directly using the UDP sockets interface (for which we have given semantics in [SSW01a, NSW02]) or using an RPC library that we have written above that. The latter would then itself require a semantic description.

The per-node code exports two interfaces – one the locally-visible API provided to users of Chord, and one the remotely-visible API invoked by other Chord nodes. Both involve abstract and concrete types as below.

```
type chordID
type ip
type port
type key = char list
type address = ip * port
type node = chordID * address

exception NotHere
```

It imports a communication interface (of one of the two forms above) which we do not develop further here.

The local API is as follows:

```
(* create a new Chord ring *)
(* raises NotHere if address not usable here *)
val create : address -> node
```

```

(* join a Chord ring containing node n' *)
val join : address * node -> node

(* find the IP address of the node responsible for this key *)
val lookup : key -> address

(* leave a Chord ring gracefully *)
val leave : unit -> unit

```

and the remote API is:

```

(* finds the successor of a chordID, by chord hops *)
val find_successor : chordID -> node

(* returns our current idea of who our predecessor is, if any *)
val predecessor : unit -> node option

(* some other node thinks it might be our predecessor *)
val notify : node -> unit

```

We can now describe the form of the semantics of the entire system. It should be a class of labelled transition systems that is appropriate for expressing two LTSs: the *implementation LTS*, defining the behaviour of the implementation, with a parallel component for each running node, and the *specification LTS*, defining a high-level specification, that captures what ‘correct’ lookups should do. Both should have the same interface.

Letting  $a$  range over machine identifiers, for each  $f : T \rightarrow T'$  in the interface above we have *labels*  $a : f!v$  and  $a : ?v$  for invocations and returns of the interface calls, where  $v$  ranges over suitably-typed values of the term language. We also have transitions labelled  $\tau$  for internal moves.

The simplest interesting model neglects communication failure and partition, but includes crash-fail of machines. We would therefore have additional labels  $a : fail$ .

The implementation LTS will also have synchronisation transitions between individual machines and the network, modelling the underlying asynchronous communication – the complete implementation LTS would be a parallel composition of a network and of a component for each machine, each of which would be a copy of the automaton specified by the MiniCaml semantics [SSW01b] for the MiniCaml implementation for the above API, thus making precise the original pseudocode. The following additional functions would be required locally, but not in the API:

```

val hash_address : address -> chordID
val hash_key : key -> chordID

(* The following three
   should get called periodically for maintenance. *)

(* verifies our immediate successor, and tells them about us *)
val stabilize : unit -> unit

(* refreshes finger table entries *)
val fix_fingers : unit -> unit

```

```
(* checks whether predecessor has failed *)
val check_predecessor : unit -> unit

(* search the local table for the highest predecessor of id *)
val closest_preceding_node : chordID -> node
```

In contrast, the specification LTS should be as simple as possible while still capturing the required behaviour. Accordingly, it should be expressed in terms of a global state. In our Consensus exercise, the global state consisted of a precise condensed account of the history of the current state. Here, the global state should contain sufficient information about the Chord ring structure, plus some information about which nodes are currently active, which nodes are supposed to be active but have crashed, and then likely accompanied with an account of all the messages that are currently pending in the overall system.

As both LTSs would be designed to have the same external labels, correctness properties could then be expressed by relating them using trace or bisimulation equivalences.

As the implementation LTS is expressed in MiniCaml, which is a sublanguage of OCaml, and as its communication is expressed using the MiniCaml sockets API (which has been implemented as a thin layer above the standard sockets library), the implementation would also be executable, supporting testing and prototyping of variants.

Returning to the discussion of applicable foundations (§3), the first step of this would be an interesting exercise in expressing key P2P distributed algorithms precisely, without pseudocode or undue abstraction. The second step would be a non-trivial verification of correctness properties, and involve specification of Chord properties in the presence of failure. The real subtlety of many P2P systems, however, is not present until one comes to consider the verification of probabilistic properties. One would like to know under ‘reasonable’ assumptions that an algorithm stabilizes ‘quickly’, with ‘high probability’, without ‘undue’ network traffic. Making these notions precise, combining the mathematical structure with the language and network semantics outlined above, and carrying out fully-formal proof, remains a long-term challenge.

## 6 Example: Analysis of Anonymity Systems

Anonymous communication is an important building block towards building privacy-aware internet applications as well as electronic voting, anonymous surveys, etc. There have been several considerable efforts to build both email-based anonymity systems (Mixmaster, Mixminion) and low-latency connection based systems (Onion Routing, Tarzan, JAP) suitable for web browsing. Both of these can be built on a P2P or a classic client-server architectures. The third class of anonymity systems is the so called P2P content sharing systems which claim to provide anonymity (Freenet and GNUnet being the prime examples).

And yet despite a variety of designs and implementations of anonymity systems, their analysis has lagged behind. Part of the reason is the difficulty of analysing anonymity properties of these systems which differ significantly from properties analysed in, for instance, the programming language or the security protocols communities.

Anonymity systems have ‘static’ and ‘dynamic’ properties. The former includes properties such as bitwise unlinkability which ensures that the attacker is unable to trivially correlate messages in different parts of the system and thus break anonymity. The latter essentially ensures that even if the attacker observes various messages going through the system, he is unable to make use of timing

information to track messages through the system. It is the dynamic properties –essentially statistical– which prove difficult to reason about.

First of all note that a message can only be anonymous within a set of messages. Thus, the job of the attacker is to decide which of the several thousand messages corresponds to the target message. This is in sharp contrast with, for instance, the case of block ciphers, where the attacker’s job is to find the correct key out of a possible  $2^{64}$  or  $2^{128}$ .

Secondly, the timing information that the attacker possesses inevitably leads to him trying to derive a probability distributions over which of the several thousand messages he sees corresponds to the target message. The inherently probabilistic nature of the analysis makes the techniques developed by the security protocols community inappropriate here.

Finally, the fact that an anonymity system is inherently open to the attackers as well as honest users leads to the necessity of dealing with active attacks. In the extreme, the active attacker is able to run a denial of service attack on everyone but one user and thus compromise their anonymity. Although this attack can be made more difficult, it always remains possible, hence the analyses which we might possibly hope to run become quantitative rather than leading to yes/no answers.

The priority in analysing anonymity systems is in working out their dynamic properties. However, there are static properties to be verified too. In particular, some of the recent anonymity systems such as Mixminion have complicated cryptographic protocols which ensure bitwise unlinkability. It would be beneficial if these are verified; this may be possible using techniques from the security protocols community such as the spi calculus.

Yet another area of important research is the development of threat models and describing them formally. It may well be the case that the standard threat models are inappropriate for a certain class of the systems, in which case weaker models would have to be created. Rigorous and precise specification would be required for an analysis to be based on these.

The first step which we have taken towards the development of the formal models was to firmly establish the probabilistic nature of the dynamic aspects of anonymity systems and develop a metric using which they can be compared [SD02]. Since, we have been working on developing models of mix networks, active attacks, real time connection-based systems and others.

## 7 Relations with Other Work Packages

- WP4**   ▷ When first versions of both are completed, we will contrast the experiences of the WP1 UDP/TCP failure semantics and the practical work in WP4 on the low layers of the DSS (which is just above the TCP sockets interface). The two should highlight different problematic areas of the protocol (and API) behaviour. Further, the low level of the DSS will be a candidate for formal work in the future.
  - ▷ The semantic work we envisage for Chord/DKS will require our UDP/TCP failure semantics.
  - ▷ Our work on dynamic binding on marshalling will feed in to the design of dynamic re-binding mechanisms for the DSS, clarifying the design options.
- WP3**   ▷ The WP1 work on versioning, modularity, and type-safe marshalling has become tightly integrated with work of WP3 on resource control (as stated elsewhere, we will report on this combined activity in future as part of WP3).
- WP2**   ▷ Work on semantics of failure detectors may be taken up in WP2 algorithms

- ▷ For work on transactions & transactional protocols, discussion is left to Deliverable D1.3.
- ▷ The techniques we have developed for reasoning about Consensus will be applied in our examination of Chord/DKS
- ▷ Some of our work on Anonymity, though reported on in WP1, has essentially been algorithm design.

## 8 Interim Progress

This document is not a progress report – that role for Work Package 1 is taken by Deliverables 1.7, 1.8, and 1.9 (for each year of the project). Nonetheless we take this opportunity to mention some of the associated publications produced since Deliverables 1.7 and 3.3. Full discussion, and the actual papers, will be included in Deliverables 1.8 and 3.4, though we include web pointers below.

**BHS<sup>+</sup>03** Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoye, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In *Proc. ICFP 2003, International Conference on Functional Programming*, August 2003. To appear. Available <http://www.cl.cam.ac.uk/users/pes20/>.

This paper develops a core theory of dynamic rebinding, and applies it to both dynamic rebinding for marshalled values and dynamic update of code.

**BHSS03** Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoye. Formalizing dynamic software updating. In *Proceedings of USE 2003: the Second International Workshop on Unanticipated Software Evolution*, April 2003. Available <http://www.cl.cam.ac.uk/users/pes20/>.

This paper develops a more specific theory of dynamic code update in the style of Erlang.

**LPSW03** James Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. ICFP 2003, International Conference on Functional Programming*, August 2003. To appear. Available <http://pauillac.inria.fr/~leifer/research.html>.

This paper develops a type- and abstraction-safe module system for marshalling values, including abstract types.

**NFM03** Uwe Nestmann, Rachele Fuzzati, and Massimo Merro. Modeling consensus in a process calculus. In *Proc. CONCUR 2003*, September 2003. To appear. <http://lamp.epfl.ch/~uwe/>.

This paper gives a precise process-calculus formalization of the Chandra-Toueg consensus algorithm, proving properties thereof with an operational model of failure and failure detection.

**OCRZ03** Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, 2003. <http://lampwww.epfl.ch/~odersky/papers/ecoop03.html>.

This paper designs and study nuObj, a calculus and dependent type system for objects and classes which can have types as members. Type members can be aliases, abstract types, or new types. The type system can model the essential concepts of Java's inner classes as well as virtual types and family polymorphism found in BETA or gbeta. It can also model most concepts

of SML-style module systems, including sharing constraints and higher-order functors, but excluding applicative functors. The type system can thus be used as a basis for unifying concepts that so far existed in parallel in advanced object systems and in module systems. The paper presents results on confluence of the calculus, soundness of the type system, and undecidability of type checking.

## 9 Conclusion

We conclude this report by summarising the foundational work required for robust P2P systems. The problems we have discussed fall into two main strands.

In the first, there are many issues concerned with the *dynamic complexity* of P2P systems: the complex algorithms involved, and the rich semantics properties we would like them to guarantee. Mathematical rigor can help here by providing techniques for specification, semantically-founded rapid prototyping, and verification – all complementing the traditional implement-and-test process, providing tools for understanding system behaviour.

In the second, work is required on the design of *high-level programming languages* for global computation. Such languages should abstract from as many distributed low-level details as possible (but no more!), enabling system builders to focus on their key architectural and algorithmic questions.

In both strands there is a problem of scale. The state of the art suffices for verification of isolated algorithms (though a single algorithm may require substantial effort), but even specification of a whole system is still a challenge. Work on programming language semantics routinely involves proof of properties about certain aspects, but a full-scale language design and implementation is likewise a large investment. The iterative development of lightweight idioms is needed in both cases, together with further use and development of mechanical proof assistants.

This report is titled “Required Foundations for Peer-to-Peer Systems”. While – obviously – successful peer-to-peer systems *can* be built with today’s technology, by sufficiently-skilled designers and programmers, both these strands of foundational work are required to make it easier to build truly robust systems, which have well-understood behaviour.

## References

- [BHS<sup>+</sup>03] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoye, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In *Proc. ICFP 2003*, August 2003. To appear.
- [BHSS03] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoye. Formalizing dynamic software updating. In *Proceedings of USE 2003: the Second International Workshop on Unanticipated Software Evolution*, April 2003.
- [JoC] JoCaml. <http://pauillac.inria.fr/jocaml/>.
- [LPSW03] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. Technical report, INRIA Rocquencourt, 2003. To appear. Draft available <http://pauillac.inria.fr/~leifer/research.html>. Also published as UCAM-CL-TR-569.

- [NSW02] Michael Norrish, Peter Sewell, and Keith Wansbrough. Rigour is good for you *and* feasible: reflections on formal treatments of C and UDP sockets. In *Tenth ACM SIGOPS European Workshop: Can We Really Depend on an OS?*, Saint-Emilion, France, September 2002.
- [Ray03] Eric S. Raymond. *The Art of Unix Programming*. Addison-Wesley, August 2003. ISBN 0-13-142901-9. Forthcoming.
- [SD02] Andrei Serjantov and George Danezis. Towards an information theoretic metric for anonymity. In Paul Syverson and Roger Dingledine, editors, *Privacy Enhancing Technologies*, volume 2482 of *Lecture Notes in Computer Science*, San Francisco, CA, April 2002. <http://petworkshop.org/2002/program.html>.
- [SMK<sup>+</sup>01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.
- [SMLN<sup>+</sup>03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.
- [SSW01a] Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP calculus: Rigorous semantics for real networking. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium (TACS 2001)*, Sendai, Japan, number 2215 in *Lecture Notes in Computer Science*, pages 535–559. Springer, October 2001.
- [SSW01b] Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP calculus: Rigorous semantics for real networking. Technical Report 515, University of Cambridge, England, July 2001.
- [SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, 1999.