Verifying a Structured Peer-to-Peer Overlay Network: The Static Case[⋆]

Johannes Borgström[1], Uwe Nestmann[1], Luc Onana[2,3], and Dilian Gurov[2,3]

[1] School of Computer and Communication Sciences, EPFL, Switzerland
[2] Department of Microelectronics and Information Technology, KTH, Sweden
[3] SICS, Sweden

Abstract. Structured peer-to-peer overlay networks are 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 the foundation for studies of a more dynamic system.

We identify a value and expression language for a value-passing CCS 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.

1 Introduction

In recent years, decentralised structured peer-to-peer (p2p) overlay networks [OEBH03, SMK+01, RD01, RFH+01] have emerged as a suitable infrastructure for scalable and robust Internet applications. However, 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.

[⋆] Supported by the EU-project IST-2001-33234 PEPITO (http://www.sics.se/pepito), part of the FET-initiative *Global Computing.*

The DKS System

In the context of the EU-project PEPITO, one of the authors is developing a decentralised structured peer-to-peer overlay network called DKS (named after the routing principle *distributed k-ary search*), of which the preliminary design can be found in [OEBH03]. DKS builds upon the idea of *relative division* [OGEA$^+$03] 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 [SMK$^+$01], Pastry [RD01] and Can [RFH$^+$01].

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.

Verification Approach

In this paper, we present 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.

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 proofs are found in the long version of the paper, which is accessible through http://lamp.epfl.ch/pepito.

Paper Overview

In Section 2 we provide a brief description of the DKS lookup algorithm, and identify the data types and functions used therein. In Section 3, we introduce a process calculus that is suitable for the description of DHT algorithms. More precisely, we may both specify and implement a DKS-based DHT in this calculus, as we do in Section 4. Finally, in Section 5 we formally prove that DKS allows to correctly implement the lookup function of DHTs by establishing a bisimulation containing the given specification and implementation.

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 [BH00] and a fault-tolerant consensus protocol [NFM03]. 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 [LT98] have been used to verify traditional (i.e., logically fully connected) distributed systems; we are not aware, though, of any p2p-examples.

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.

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.

2 DKS

In this section we briefly describe the DKS system, focusing on the lookup algorithm. More information about the DKS system can be found for instance in [OEBH03, OGEA+03].

For the design of the DKS system, we model a distributed system as a set of *processes* linked together through a *communication network*. Processes communicate by message passing and a process reacts upon receipt of a message; i.e., this is an event-driven model. The communication network is assumed to be (*i*) *connected*, each process can send a message directly to any other process in the system; (*ii*) *asynchronous*, the time taken by the communication network to forward a message to its destination can be arbitrarily long; (*iii*) *reliable*, messages are neither lost nor duplicated.

2.1 The Virtual Identifier Space

For DKS, as for other structured peer-to-peer overlay networks [SMK+01, RD01], participating nodes are uniquely identified by identifiers from a set called *identifier space*. As in Chord and Pastry, the identifier space for DKS is a ring of size N that we identify with $\mathbb{Z}_N$, where we write $\mathbb{Z}_n$ for $\{0, 1, \cdots, n-1\}$. To model the ring structure, we let $\oplus$ and $\ominus$ be addition and subtraction modulo N, with the convention that the results of modular arithmetic are always non-negative and strictly less than the modulus. For simplicity, it is assumed that $N = k^d$ for $k > 1$, $d > 1$, where k will be the *branching factor* of the search tree. We work with a static system, with a fixed set of participating nodes $\mathcal{I} \subseteq \mathbb{Z}_N$ with $|\mathcal{I}| > 1$.

2.2 Assignment of Key-Value Pairs to Nodes

As part of the specification of a DHT, we assume that data items to be stored into and retrieved from the system are pairs $(key, val) \in \mathbb{N} \times \mathbb{N}$ where the keys are assumed to be unique. We model the data items currently in the system as a partial function data : $\mathbb{N} \rightharpoonup \mathbb{N}$. Using some arbitrary hashing function, $\mathrm{H} : \mathbb{N} \to \mathbb{Z}_N$, the *key* of a data item is hashed to obtain a *key identifier* $\mathrm{H}(key)$ for the pair (key, val).

In DKS (as well as in Chord), a data item (key, val) is stored at the first node succeeding $\mathrm{H}(key)$. That node is called the *successor* of $\mathrm{H}(key)$, and is defined as $\mathrm{suc}(i) \in \{j \in \mathcal{I} \mid j \ominus i = \min\{h \ominus i \mid h \in \mathcal{I}\}\}$. Note that $\mathrm{suc}(\cdot)$ is well-defined since $h \ominus i = j \ominus i$ iff $h \ominus j = 0$. Dually, the (strict) predecessor of a node $i \in \mathcal{I}$ is $\mathrm{pre}(i) \in \{j \in \mathcal{I} \mid j \ominus i = \max\{h \ominus i \mid h \in \mathcal{I}\}\}$. Local lookup at node n is a partial function $\mathrm{data}_n(j) := \mathrm{data}(j)$ if $\mathrm{suc}(j) = n$, i.e., returning the value $\mathrm{data}(j)$ associated to a key j only on the node n responsible for the item (key, val).

2.3 Routing Tables

The DKS system is built in a way that allows any node to reach any other node in *at most* $\log_k(N)$ hops under normal system operation. To achieve this, the principle of *relative division* of the space [OGEA+03] is used to embed, at each point of the identifier space, a complete virtual k-ary tree of height $d = \log_k(N)$. We let $\mathcal{L} := \{1, 2, \cdots, d\}$ be the levels of this tree, where 1 is the top level (the root). At a level $l \in \mathcal{L}$, a node n has a *view* V^l of the identifier space. The view V^l consists of k equal parts, denoted I^l_i, $0 \leq i \leq k-1$, and defined below level by level.

At level 1: $V^1 = I^1_0 \uplus I^1_1 \uplus I^1_2 \uplus \cdots \uplus I^1_{k-1}$, where $I^1_0 = [x^1_0, x^1_1)$, $I^1_1 = [x^1_1, x^1_2)$, $\cdots$, $I^1_{k-1} = [x^1_{k-1}, x^1_0)$, $x^1_i = n \oplus i\frac{N}{k}$, for $0 \leq i \leq k-1$.

At level $2 \leq l \leq d$: $V^l = I^l_0 \uplus I^l_1 \uplus I^l_2 \uplus \cdots \uplus I^l_{k-1}$, where $I^l_0 = [x^l_0, x^l_1)$, $I^l_1 = [x^l_1, x^l_2)$, $\cdots$, $I^l_{k-1} = [x^l_{k-1}, x^{l-1}_1)$, $x^l_i = n \oplus i\frac{N}{k^l}$, for $0 \leq i \leq k-1$.

To construct the routing table, denoted Rt_n, of an arbitrary node n of a DKS system we take for each level $l \in \mathcal{L}$ and each interval i at level l a pointer to the *successor* of x^l_i, as defined above.

Routing Table Example. As an example, consider an identifier space of size $N = 4^2$, i.e., $d = 2$ and $k = 4$. Assume that the nodes in the system are $\mathcal{I} := \{0, 2, 5, 10, 13\}$. In this case, using the principle described above for building routing table in DKS, we have that node 0 has the routing table in Figure 1.

Level	Interval	Responsible	Level	Interval	Responsible
1	$[0,4)$	0	2	$[0,1)$	0
	$[4,8)$	5		$[1,2)$	2
	$[8,12)$	10		$[2,3)$	2
	$[12,0)$	13		$[3,4)$	5

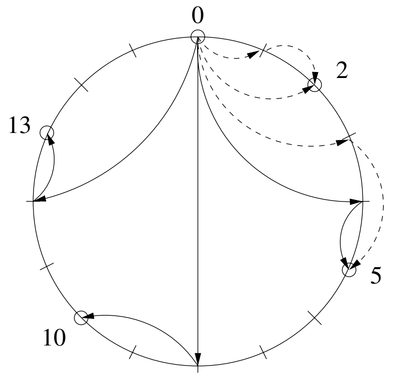

Fig. 1. Routing table for node 0

Formally, the routing tables of the nodes are partial functions

$$\mathrm{Rt}_n(j,l) := \mathrm{suc}\left(n \oplus \left(\frac{N}{k^l}\left\lfloor \frac{(j \ominus n)k^l}{N} \right\rfloor\right)\right) \text{ if } j \ominus n < k^{d+1-l} \text{ and } l \leq d,$$

where $\mathrm{Rt}_n(j,l)$ is the node responsible for the interval containing j on level l according to node n. We also define the lookup level for an identifier at a given node as $\mathrm{lvl}_n(j) := d - \lfloor \log_k(j \ominus n) \rfloor$, and let lookup in the routing table be $\mathrm{Rt}_n(j) := \mathrm{Rt}_n(j, \mathrm{lvl}_n(j))$, which is defined for all n, j.

2.4 Lookup in a Static DKS

The specification of lookup is common to all DHTs: A lookup for a key *key* at a node n should simply return the associated data value (if any) to the user on node n. Moreover, the system should always be available for new requests, and the responses may be returned in any order.

In DKS, the lookup can be done either iteratively, transitively or recursively. These are well-known strategies for resolving names in distributed systems [Gos91]. In this paper, we present a simplified version of the *recursive* algorithm of DKS.

Briefly and informally, the recursive lookup in the DKS system goes as follows. When a DKS node n receives a request for a key *key* from its user, u, node n checks if the virtual identifier associated to *key* is between $\mathrm{pre}(n)$ and n. If so, node n performs a local lookup and returns the value associated to *key* to the user. Otherwise, node n starts forwarding the request, such that it descends through the virtual k-ary tree associated with node n until the unique node z

such that $\mathrm{H}(key)$ is between $\mathrm{pre}(z)$ and z is reached. We call z the manager of *key*.

When the manager of *key* is reached, it does a local lookup to determine the value associated with *key*. This value is returned, back-tracing the path taken by the request. In order to do this, a stack is embedded in each internal request message, such that at each step of the forwarding process, the node n' handling the message pushes itself onto the stack. The manager z then starts a "forwarding" of internal response messages towards the origin of the request. Each such message carries the result of the lookup as well as the stack.

When a node n receives an internal response message, node n checks if the stack attached to the message is empty. If not, the head of the stack determines the next step in the "backwarding" of the message towards its origin. If the stack is empty, then n was the origin of the lookup. Then node n returns the result of the response to its user, u.

The back-tracing makes the response follow a "trusted path", to route around possible link failures, e.g., between the manager of the key and the originator of the lookup. The stack also provides some fault-tolerance: If the node at the head of the stack is no longer reachable, the nodes below can be used to return the message.

A formal model of this lookup algorithm can be found in Section 4, using the process calculus defined in Section 3.

3 Language

We use a variant of value-passing CCS [Mil89, Ing94] to implement the DKS system described above. To separate unrelated features and allow for a simple adaptation to the verification of other algorithms, we clearly distinguish three orthogonal aspects of the calculus.

Values and Expressions: The values $\mathcal{V}$ are integers, lists in nil $[\,]$ and cons $v_1 :: v_2$ format and the "undefined value" $\perp$. The expressions $\mathcal{E}$ contain some standard operations on values, plus common DHT functions and DKS-specific functions seen in Section 2.

We extend the domain and codomain of $F \in \{\mathrm{data}, \mathrm{lvl}_v, \mathrm{data}_v, \mathrm{Rt}_v \mid v \in \mathcal{I}\}$ to $\mathcal{V}$ by letting $F(v) := \perp$ for the values v on which F was previously undefined. We extend the domain of H to $\mathcal{V}$ by letting H take arbitrary values in $\mathbb{Z}_N$ for values not in $\mathbb{N}$. Expressions are evaluated using the function $[\![\cdot]\!] : \mathcal{E} \to \mathcal{V}$.

For boolean checks $\mathcal{B}$, we have the matching construct $e_1 = e_2$ and an interval check $e_1 \in (e_2, e_3\,]$ modulo N. Boolean checks are evaluated using the predicate $\mathbf{e}_\mathrm{b}(\cdot)$. Values and boolean checks are defined in Table 1, both $[\![\cdot]\!]$ and $\mathbf{e}_\mathrm{b}(\cdot)$ are defined in Table 2. We do not use a typed value language, although the equivalence result obtained in Section 5.2 intuitively implies that the implementation is "as well-typed as" the specification.

We use tuples $\tilde{e}$ of expressions (and other terms), where $\tilde{e} := e_1, \ldots, e_{|\tilde{e}|}$ that may be empty, i.e., $|\tilde{e}| = 0$. To evaluate a tuple of expressions, we write $\mathrm{J}\tilde{e}\mathrm{K}$ for the tuple of values $\mathrm{J}e_1\mathrm{K}, \ldots, \mathrm{J}e_{|\tilde{e}|}\mathrm{K}$.

As a more compact representation of lists of values, we write $[u\tilde{v}]$ for $u :: [\tilde{v}]$, and also define $\mathrm{last}([v_1, v_2, \cdots, v_n]) := v_n$ if $n > 0$.

Parallel Language: We use a polyadic value-passing CCS, with asynchronous output and input-guarded choice. We assume that the set of names $a, b \in \mathcal{N}$ and the set of variables $x, y \in \mathcal{W}$ are disjoint and infinite. The syntax of the calculus can be found in Table 1.

As an abbreviation we write $\sum_{j\in\mathcal{J}} G_j$ for $\mathbf{0} + G_{j_0} + G_{j_1} + \cdots + G_{j_n}$ and $\prod_{j\in\mathcal{J}} P_j$ for $\mathbf{0} \,|\, P_{j_0} \,|\, P_{j_1} \,|\, \cdots \,|\, P_{j_n}$, where $\mathcal{J} = \{j_i \mid 0 \leq i \leq n\}$ ($\mathcal{J}$ may be $\emptyset$).

Control Flow Structures: We use the standard **if** ϕ **then** P **else** Q and a switch statement **case** e **of** $\{j \mapsto P_j \mid j \in S\}$ for a more compact representation of nested comparisons of the same value. In all **case** statements, we require $S \subset \mathcal{V}$ to be finite.

To gain a closer correspondence to the method-oriented style usually used when presenting distributed algorithms, we work with defining equations for process constants $\mathsf{A}\langle\tilde{e}\rangle$ rather than recursive definitions embedded in the process terms. If a process constant A does not take any parameters, we write A for both $\mathsf{A}\langle\rangle$ and $\mathsf{A}()$.

Table 1. Syntax

u, v	$::= 0, 1, 2, \cdots \;\mid\; [\,] \;\mid\; \bot \;\mid\; u :: u$	values $\mathcal{V}$
e	$::= u \;\mid\; x$	expressions $\mathcal{E}$
	$\mid\; \mathsf{head}(e) \;\mid\; \mathsf{tail}(e) \;\mid\; e :: e$	(lists)
	$\mid\; \mathsf{data}(e) \;\mid\; \mathsf{H}(e)$	(global)
	$\mid\; \mathsf{lvl}_v(e) \;\mid\; \mathsf{data}_v(e) \;\mid\; \mathsf{Rt}_v(e)$	(local)
ϕ, ψ	$::= e = e$	boolean tests $\mathcal{B}$
	$\mid\; e \in (e, e]$	(interval check)
G	$::= \mathbf{0}$	input-guarded sums $\mathcal{G}$
	$\mid\; a(\tilde{x}).P$	(input prefix)
	$\mid\; G + G$	(choice)
P, Q	$::= G$	processes $\mathcal{P}$
	$\mid\; \overline{a}\langle\tilde{e}\rangle$	(asynchronous output)
	$\mid\; P \,\|\, P$	(parallel)
	$\mid\; (P) \setminus a$	(restriction)
	$\mid\; \mathsf{A}\langle\tilde{e}\rangle$	(process constant)
	$\mid\; \mathbf{if}\ \phi\ \mathbf{then}\ P\ \mathbf{else}\ P$	(if statement)
	$\mid\; \mathbf{case}\ e\ \mathbf{of}\ \{j \mapsto P_j \mid j \in S\}$	(case statement)

3.1 Semantics

The set of actions $\mathcal{A} \ni \mu$ is defined as $\mu ::= \tau \,|\, a\,\tilde{v} \,|\, \overline{a}\,\tilde{v}$. The channel of an action, $\mathrm{ch} : \mathcal{A} \to \mathcal{N} \cup \{\bot\}$, is defined as $\mathrm{ch}(\tau) := \bot$, $\mathrm{ch}(a\,\tilde{v}) := a$ and $\mathrm{ch}(\overline{a}\,\tilde{v}) := a$. The variables $\tilde{x}$ are bound in $a(\tilde{x}).P$. Substitution of the values $\tilde{v}$ for the variables $\tilde{x}$ in process P is written $P\,[^{v_1}/_{x_1}, \ldots, ^{v_n}/_{x_n}]$ and performed recursively on the non-bound instances of $\tilde{x}$ in P. We use a standard labelled structural operational semantics with early input (see Table 2). To compute the values to be transmitted, instantiate process constants and evaluate **if** and **case** statements we use an auxiliary reduction relation $>$ (see Table 2).

Structural congruence is a standard notion of equivalence (cf. [MPW92]) that identifies process terms based on their syntactic structure. In a value-passing language, it often includes simplifications resulting from the evaluation of "top-level" expressions (cf. [AG99]). In our calculus, top-level evaluation is treated by the reduction relation $>$, which is contained in the structural congruence.

Definition 1 (Structural Congruence). Structural congruence $\equiv$ *is the least equivalence relation on* $\mathcal{P}$ *containing* $>$ *and satisfying commutative monoid laws for* $(\mathcal{P}, |\,, 0)$ *and* $(\mathcal{G}, +, 0)$ *and the following inference rules.*

$$\textsc{S-par}\ \frac{P_1 \equiv P_1'}{P_1 \,|\, P_2 \equiv P_1' \,|\, P_2} \qquad \textsc{S-sum}\ \frac{G_1 \equiv G_1'}{G_1 + G_2 \equiv G_1' + G_2} \qquad \textsc{S-res}\ \frac{P \equiv P'}{(P) \setminus a \equiv (P') \setminus a}$$

Depending on the actual structural congruence rules at hand it is well known, and can easily be shown, that structurally congruent processes give rise to the "same" transitions (leading to again structurally congruent processes) according to the operational semantics. Thus, transitions can be seen as a relation between congruence classes of processes. To simplify descriptions of the behaviour of processes, we define a related notion where we instead work with representatives for the congruence classes.

Definition 2 (Transition Graph Up to Structural Congruence). *A* transition graph up to structural congruence *is a labelled relation* $\Rrightarrow \subseteq \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$ *for* $\mathcal{Q} \subseteq \mathcal{P}$ *such that for all* $Q \in \mathcal{Q}$ *we have that*

– *If* $Q \xrightarrow{\mu} P'$*, there is* Q' *such that* $Q \overset{\mu}{\Rrightarrow} Q'$ *and* $P' \equiv Q'$.
– *If* $Q \overset{\mu}{\Rrightarrow} Q'$*, there is* P' *such that* $Q \xrightarrow{\mu} P'$ *and* $P' \equiv Q'$.

We say that $\Rrightarrow$ *is a transition graph up to* $\equiv$ *for* Q *if* $Q \in \mathcal{Q}$.

According to this definition, it is sufficient to include just one representative for the congruence class of a derivative; however, one may include several.

Weak bisimulation is a standard equivalence [Mil89] identifying processes with the same externally observable reactive behaviour, ignoring invisible internal activity. We define this process equivalence with respect to a general labelled transition system; this allows us to interpret the notion also on transition graphs up to $\equiv$.

Table 2. Semantics

Expression evaluation and boolean evaluation are defined as follows:

$$[\![e]\!] := \begin{cases} v & \text{if } e = v \in \mathcal{V} \\ v_1 & \text{if } e = \mathsf{head}(e') \text{ and } [\![e']\!] = v_1 :: v_2 \\ v_2 & \text{if } e = \mathsf{tail}(e') \text{ and } [\![e']\!] = v_1 :: v_2 \\ v_1 :: v_2 & \text{if } e = e_1 :: e_2 \text{ and } [\![e_1]\!] = v_1, [\![e_2]\!] = v_2 \\ \mathrm{F}([\![e']\!]) & \text{if } e = \mathsf{F}(e') \text{ and } \mathsf{F} \in \{\mathsf{data}, \mathsf{H}, \mathsf{lvl}_v, \mathsf{data}_v, \mathsf{Rt}_v \mid v \in \mathcal{I}\} \\ \bot & \text{if otherwise} \end{cases}$$

$$\mathbf{e}_\mathrm{b}(\, e_1 = e_2 \,) \text{ is true iff } [\![e_1]\!] = [\![e_2]\!] \neq \bot$$
$$\mathbf{e}_\mathrm{b}(\, e_1 \in (e_2, e_3\,]\,) \text{ is true iff } [\![e_i]\!] = n_i \in \mathbb{N} \text{ for } i \in \{1,2,3\} \text{ and } 0 < n_1 \ominus n_2 \leq n_3 \ominus n_2$$

The (top-level) *reduction relation* $>$ is the least relation on $\mathcal{P}$ satisfying:

1. $\overline{a}\langle\tilde{e}\rangle > \overline{a}\langle\tilde{v}\rangle$ if $[\![\tilde{e}]\!] = \tilde{v}$.
2. $\mathsf{A}\langle\tilde{e}\rangle > P\,[{}^{v_1}/_{x_1}, \ldots, {}^{v_n}/_{x_n}]$ if $\mathsf{A}(\tilde{x}) \overset{def}{=} P$, $|\tilde{e}| = |\tilde{x}| = n$ and $[\![\tilde{e}]\!] = \tilde{v}$.
3. **if** ϕ **then** P **else** Q $> P$ if $\mathbf{e}_\mathrm{b}(\phi)$.
4. **if** ϕ **then** P **else** Q $> Q$ if $\neg\,\mathbf{e}_\mathrm{b}(\phi)$.
5. **case** e **of** $\{j \mapsto P_j \mid j \in S\} > P_v$ if $[\![e]\!] = v \in S$.

The structural operational semantics are given by the following inference rules, where the symmetric versions of Com-L, Par-L and Sum-L have been omitted.

$$(\textsc{in})\ \frac{}{a(\tilde{x}).P \xrightarrow{a\,\tilde{v}} P\,[{}^{v_1}/_{x_1}, \ldots, {}^{v_n}/_{x_n}]} \text{ if } |\tilde{v}| = |\tilde{x}| \qquad (\textsc{out})\ \frac{}{\overline{a}\langle\tilde{v}\rangle \xrightarrow{\overline{a}\,\tilde{v}} \mathbf{0}}$$

$$(\textsc{com-L})\ \frac{P \xrightarrow{a\,\tilde{v}} P' \qquad Q \xrightarrow{\overline{a}\,\tilde{v}} Q'}{P \,|\, Q \xrightarrow{\tau} P' \,|\, Q'} \qquad (\textsc{par-L})\ \frac{P \xrightarrow{\mu} P'}{P \,|\, Q \xrightarrow{\mu} P' \,|\, Q}$$

$$(\textsc{sum-L})\ \frac{G_1 \xrightarrow{a\,\tilde{v}} P'}{G_1 + G_2 \xrightarrow{a\,\tilde{v}} P'} \qquad (\textsc{res})\ \frac{P \xrightarrow{\mu} P'}{(P) \setminus a \xrightarrow{\mu} (P') \setminus a} \text{ if } a \neq \mathrm{ch}(\mu)$$

$$(\textsc{red})\ \frac{P > Q \qquad Q \xrightarrow{\mu} Q'}{P \xrightarrow{\mu} Q'}$$

Definition 3 (Weak Bisimulation). *If* $\rightsquigarrow \subseteq \mathcal{P}\times\mathcal{A}\times\mathcal{P}$ *then a binary relation* $\mathcal{S} \subseteq \mathcal{P}\times\mathcal{P}$ *is a weak* $\rightsquigarrow$*-bisimulation if*
whenever $P \mathrel{\mathcal{S}} Q$ *and* $P \overset{\mu}{\rightsquigarrow} P'$ *there exists* Q' *such that* $P' \mathrel{\mathcal{S}} Q'$ *and*

- *if* $\mu = \tau$ *then* $Q \overset{\tau}{\rightsquigarrow}{}^* Q'$*;*
- *if* $\mu \neq \tau$ *then* $Q (\overset{\tau}{\rightsquigarrow}{}^*) \overset{\mu}{\rightsquigarrow} (\overset{\tau}{\rightsquigarrow}{}^*) Q'$*,*

and conversely for the transitions of Q*.*

The notion usually deployed in process calculi is weak $\rightarrow$-bisimilarity: P is *weakly $\rightarrow$-bisimilar* to Q, written $P \approx Q$, if there is a weak $\rightarrow$-bisimulation $\mathcal{S}$ with $P\,\mathcal{S}\,Q$[1].

Next, we use the concept of $\Rrightarrow$-bisimilarity as simple proof technique: two processes are weakly $\rightarrow$-bisimilar if they are weakly $\Rrightarrow$-bisimilar.

Proposition 1. *If $\mathcal{S}$ is a weak $\Rrightarrow$-bisimulation, then $\equiv\mathcal{S}\equiv$ is a weak $\rightarrow$-bisimulation.*

4 Specification and Implementation

We now use the process calculus defined in Section 3 to specify and implement lookup in the DKS system.

Specification. In the specification process Spec, lookup requests and results are transmitted on indexed families of names $\mathit{request}_i, \mathit{response}_i \in \mathcal{N}$, where the index corresponds to the node the channel is connected to. The $\mathit{request}_i$ channels carry a single value: the key to be looked up. The $\mathit{response}_i$ channels carry the key and the associated data value.

$$\mathsf{Spec} \stackrel{\mathit{def}}{=} \sum_{i\,\in\,\mathcal{I}} \mathit{request}_i(\mathit{key}).(\overline{\mathit{response}_i}\langle \mathit{key}, \mathsf{data}(\mathit{key})\rangle \mid \mathsf{Spec}).$$

Implementation. The process implementing the DKS system, defined in Table 3, consists of a collection of nodes. A node Node_i is a purely reactive process that receives on the associated $\mathit{request}_i$, req_i and resp_i channels, and sends on $\mathit{response}_i$, req_j and resp_j for $j \in \mathrm{range}(\mathrm{Rt}_i(\cdot))$. The req_i channels carry three values: the key to be looked up, a stack specifying the return path for the result, and the current lookup level. The resp_i channels carry the key, the found value and the remaining return path.

Requests, i.e., messages on channels $\mathit{request}_i$ and req_i, are treated by the subroutine Req_i, which decides *whether* to respond to the message directly or to route it towards its destination. This decision is naturally based on whether it is itself responsible for the key searched for, as defined in Section 2; in this case, it responds with the value of a local lookup. Responses, i.e., messages on channels resp_i, are treated by the subroutine Resp_i, which decides *to whom* precisely to pass on the response; depending on the call stack, it either returns itself the result of a query to the application, or it passes on the response to the node from whom the request arrived earlier on.

The implementation of the static DKS system, Impl, is then simply the parallel composition of all nodes, with a top-level restriction on the channels that are not present in the DHT API. We use variables $\mathit{key}, \mathit{stack}, \mathit{value}, \mathit{level} \in \mathcal{W}$.

[1] The knowledgeable reader may note that although we find ourselves within a calculus with asynchronous message-passing, we use a standard synchronous bisimilarity, which is known to be strictly stronger than the notion of asynchronous bisimilarity. However, our correctness result holds even for this stronger version.

Table 3. DKS Implementation

$$\begin{aligned}\mathsf{Node}_i \overset{def}{=}\ & \mathit{request}_i(\mathit{key}).(\mathsf{Node}_i \mid \mathsf{Req}_i\langle \mathit{key}, [\,]\rangle)\\ +\ & \mathit{req}_i(\mathit{key}, \mathit{stack}, \mathit{level}).(\mathsf{Node}_i \mid \mathsf{Req}_i\langle \mathit{key}, \mathit{stack}\rangle)\\ +\ & \mathit{resp}_i(\mathit{key}, \mathit{value}, \mathit{stack}).(\mathsf{Node}_i \mid \mathsf{Resp}_i\langle \mathit{key}, \mathit{value}, \mathit{stack}\rangle)\end{aligned}$$

$$\begin{aligned}\mathsf{Req}_i(\mathit{key}, \mathit{stack}) \overset{def}{=}\ & \textbf{if}\quad \mathsf{H}(\mathit{key}) \in (\mathrm{pre}(i), i\,]\\ & \textbf{then}\ \mathsf{Resp}_i\langle \mathit{key}, \mathsf{data}_i(\mathit{key}), \mathit{stack}\rangle\\ & \textbf{else}\ \ \textbf{case}\ \mathsf{Rt}_i(\mathsf{H}(\mathit{key}))\\ & \qquad\ \ \textbf{of}\quad \{j \mapsto \overline{\mathit{req}_j}\langle \mathit{key}, i :: \mathit{stack}, \mathsf{lvl}_i(\mathsf{H}(\mathit{key}))\rangle \mid j \in \mathcal{I}\}\end{aligned}$$

$$\begin{aligned}\mathsf{Resp}_i(\mathit{key}, \mathit{value}, \mathit{stack}) \overset{def}{=}\ & \textbf{if}\quad \mathit{stack} = [\,]\\ & \textbf{then}\ \overline{\mathit{response}_i}\langle \mathit{key}, \mathit{value}\rangle\\ & \textbf{else}\ \ \textbf{case}\ \mathsf{head}(\mathit{stack})\\ & \qquad\ \ \textbf{of}\quad \{j \mapsto \overline{\mathit{resp}_j}\langle \mathit{key}, \mathit{value}, \mathsf{tail}(\mathit{stack})\rangle \mid j \in \mathcal{I}\}\end{aligned}$$

$$\mathsf{Impl} \overset{def}{=} \left(\prod_{i\,\in\,\mathcal{I}} \mathsf{Node}_i\right) \setminus \{\mathit{req}_i, \mathit{resp}_i \mid i \in \mathcal{I}\}$$

5 Correctness

Our correctness result is that the specification of lookup is weakly bisimilar to its (non-diverging) implementation in the DKS system. We show this by providing a uniform representation of the derivatives of the specification and the implementation, and their transition graphs up to $\equiv$, allowing us to directly exhibit the bisimulation.

5.1 State Space and Transition Graph

Since nodes are stateless (in the static setting), we only need to keep track of the messages currently in the system. For this we will use multisets, with the following notation: A *multiset* M over a set $\mathcal{M}$ is a function with type $\mathcal{M} \to \mathbb{N}$. By $\mathrm{spt}(M) := \{x \in \mathcal{M} \mid M(x) \neq 0\}$, we denote the *support* of M. We write 0 for any multiset with empty support. We can add and remove items by $S + a := \{a \mapsto S(a){+}1\} \cup \{x \mapsto S(x) \mid x \in \mathrm{dom}(S) \setminus \{a\}\}$ when $a \in \mathrm{dom}(S)$ and $S - a := \{a \mapsto S(a){-}1\} \cup \{x \mapsto S(x) \mid x \in \mathrm{dom}(S) \setminus \{a\}\}$, where $S - a$ is defined only when $a \in \mathrm{spt}(S)$. More generally, we define the sum of two multisets with the same domain as $S + T := \{x \mapsto S(x){+}T(x) \mid x \in \mathrm{dom}(S)\}$.

Specification. The states of the lookup specification are uniquely determined by the undelivered responses. To describe this state space, we define families of process constants $\mathsf{Responses}_\alpha$ and Spec_α, where α ranges over multisets with domain $\mathcal{I} \times \mathcal{V}$ and finite support. We write $t < n$ for $t \in \mathbb{Z}_n$.

$$\mathsf{Responses}_\alpha \overset{def}{=} \prod_{(i,kv)\,\in\,\mathrm{spt}(\alpha)}\ \prod_{t<\alpha(i,kv)} \overline{\mathit{response}_i}\langle kv, \mathrm{data}(kv)\rangle$$

Let $\mathsf{Spec}_\alpha := \mathsf{Responses}_\alpha | \mathsf{Spec}$. Note that $\mathsf{Spec} \equiv \mathsf{Spec}_0$.

Lemma 1. Spec_0 *has the following transition graph up to* $\equiv$.

1. $\mathsf{Spec}_\alpha \xRightarrow{request_i\ kv} \mathsf{Spec}_{\alpha+(i,kv)}$ *if* $i \in \mathcal{I}$ *and* $kv \in \mathcal{V}$
2. $\mathsf{Spec}_\alpha \xRightarrow{\overline{response_i}\ kv, \mathrm{data}(kv)} \mathsf{Spec}_{\alpha-(i,kv)}$ *if* $(i, kv) \in \mathrm{spt}(\alpha)$

Implementation. For the implementation, we also have to keep track of $resp_i$ and req_i messages and the values that can be sent in them. Since the routing tables are correctly configured, there is a simple invariant on the parameters of the $\overline{req_i}\langle kv, L, m\rangle$ messages in the system: Such messages are either sent to the node responsible for kv, or to the node responsible for the interval containing $\mathrm{H}(kv)$ on level m as discussed in Section 2. To capture this invariant we let $list[\mathcal{I}] := \{[i_1, i_2, \cdots, i_n] \mid i_j \in \mathcal{I} \wedge n \in \mathbb{N}\}$, and define $\mathcal{R} \subset \mathcal{I} \times \mathcal{V} \times list[\mathcal{I}] \times \mathbb{Z}_{d+1}$ as

$$\mathcal{R} := \{(i, kv, L, m) \mid L \neq [\,] \wedge \\ (\ \mathrm{suc}(\mathrm{H}(kv)) = i \ \vee\ \mathbf{e}_\mathrm{b}(\,\mathrm{H}(kv) \in\ (i,\ i \oplus k^{d-m} \ominus 1\,]\,)\ )\}.$$

To model the internal messages in the DKS system, we define families of process constants Reqs_β and Resps_γ where α is as above, β ranges over multisets with domain $\mathcal{R}$ and finite support and γ ranges over multisets with domain $\mathcal{I} \times \mathcal{V} \times list[\mathcal{I}]$ and finite support as follows.

$$\mathsf{Reqs}_\beta \overset{def}{=} \prod_{(i,kv,L,m)\,\in\,\mathrm{spt}(\beta)} \ \prod_{t<\beta(i,kv,L,m)} \overline{req_i}\langle kv, L, m\rangle$$

$$\mathsf{Resps}_\gamma \overset{def}{=} \prod_{(i,kv,L)\,\in\,\mathrm{spt}(\gamma)} \ \prod_{t<\gamma(i,kv,L)} \overline{resp_i}\langle kv, \mathrm{data}(kv), L\rangle$$

The behaviour of the implementation is captured by the constants $\mathsf{Impl}_{\alpha,\beta,\gamma}$.

$$\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{def}{=} \left(\mathsf{Responses}_\alpha | \mathsf{Reqs}_\beta | \mathsf{Resps}_\gamma | \prod_{i\,\in\,\mathcal{I}} \mathsf{Node}_i\right) \setminus \{req_i, resp_i \mid i \in \mathcal{I}\}$$

Note that $\mathsf{Impl} \equiv \mathsf{Impl}_{0,0,0}$.

Lemma 2. $\mathsf{Impl}_{0,0,0}$ *has the following transition graph up to* $\equiv$.

1. $\mathsf{Impl}_{\alpha,\beta,\gamma} \xRightarrow{request_i\ kv} \mathsf{Impl}_{\alpha+(i,kv),\beta,\gamma}$
   if $i \in \mathcal{I}$ *and* $\mathbf{e}_\mathrm{b}(\,\mathrm{H}(kv) \in\ (\mathrm{pre}(i),\ i\,]\,)$
2. $\mathsf{Impl}_{\alpha,\beta,\gamma} \xRightarrow{request_i\ kv} \mathsf{Impl}_{\alpha,\beta+(\mathrm{Rt}_i(\mathrm{H}(kv)),kv,[i],\mathrm{lvl}_i(\mathrm{H}(kv))),\gamma}$
   if $i \in \mathcal{I}$ *and* $\neg\big(\ \mathbf{e}_\mathrm{b}(\,\mathrm{H}(kv) \in\ (\mathrm{pre}(i),\ i\,]\,)\big)$
3. $\mathsf{Impl}_{\alpha,\beta,\gamma} \xRightarrow{\overline{response_i}\ kv, \mathrm{data}(kv)} \mathsf{Impl}_{\alpha-(i,kv),\beta,\gamma}$
   if $(i, kv) \in \mathrm{spt}(\alpha)$

4. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha,\beta-(i,kv,h::L,m),\gamma+(h,kv,L)}$
if $(i, kv, h{::}L, m) \in \mathrm{spt}(\beta)$ *and* $\mathbf{e}_\mathrm{b}(\,\mathrm{H}(kv) \in (\mathrm{pre}(i),\, i\,]\,)$

5. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha,\beta-(i,kv,L,m)+(\mathrm{Rt}_i(\mathrm{H}(kv)),kv,i::L,\mathrm{lvl}_i(\mathrm{H}(kv))),\gamma}$
if $(i, kv, L, m) \in \mathrm{spt}(\beta)$ *and* $\neg\big(\,\mathbf{e}_\mathrm{b}(\,\mathrm{H}(kv) \in (\mathrm{pre}(i),\, i\,]\,)\big)$

6. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha,\beta,\gamma-(i,kv,h::L)+(h,kv,L)}$
if $(i, kv, h{::}L) \in \mathrm{spt}(\gamma)$

7. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha+(i,kv),\beta,\gamma-(i,kv,[\,])}$
if $(i, kv, [\,]) \in \mathrm{spt}(\gamma)$

Having found the transition graphs of both the specification and the implementation up to structural congruence, we restrict ourselves to working with this transition system.

Definition 4. *Let* $\Rrightarrow$ *be the union of the relations in the statements of Lemma 1 and Lemma 2.*

Note that $\Rrightarrow$ is as transition graph up to structural congruence for both Spec_0 and $\mathsf{Impl}_{0,0,0}$.

5.2 Bisimulation

To relate the state spaces of the specification and the implementation, we define two partial functions $\mathrm{T}_{req} : \mathcal{R} \rightharpoonup (\mathcal{I} \times \mathbb{N})$ and $\mathrm{T}_{resp} : (\mathcal{I} \times \mathbb{N} \times \mathit{list}[\mathcal{I}]) \rightharpoonup (\mathcal{I} \times \mathbb{N})$ that map the parameters of *req* and *resp* messages, respectively, to those of the corresponding *response* messages as follows.

$$\mathrm{T}_{req}(i, kv, L, m) := (\mathrm{last}(L), kv)$$
$$\mathrm{T}_{resp}(i, kv, L) := \begin{cases} (\mathrm{last}(L), kv) & \text{if } L \neq [\,] \\ (i, kv) & \text{if } L = [\,] \end{cases}$$

Note that T_{req} is well-defined since $\mathrm{dom}(\mathrm{T}_{req}) = \mathcal{R}$, thus $L \neq [\,]$. We then lift these functions to the respective multisets of type β and γ.

$$\widehat{\mathrm{T}_{req}}(\beta) := \sum_{x \in \mathrm{spt}(\beta)} \{\, \mathrm{T}_{req}(x) \mapsto \beta(x) \,\}$$
$$\widehat{\mathrm{T}_{resp}}(\gamma) := \sum_{x \in \mathrm{spt}(\gamma)} \{\, \mathrm{T}_{resp}(x) \mapsto \gamma(x) \,\}$$

Here $\sum$ denotes indexed multiset summation. Finally, we abbreviate the accumulated expected visible responses due to pending requests by:

$$\widehat{\mathrm{T}}(\alpha, \beta, \gamma) := \alpha + \widehat{\mathrm{T}_{req}}(\beta) + \widehat{\mathrm{T}_{resp}}(\gamma)$$

The implementation has a well-defined behaviour on internal transitions, as the following two lemmas show. First, internal transitions does not change the equivalence classes under the equivalence induced by the $\widehat{\mathrm{T}}$-transformation.

Lemma 3. *If* $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha',\beta',\gamma'}$ *then* $\widehat{\mathrm{T}}(\alpha,\beta,\gamma) = \widehat{\mathrm{T}}(\alpha',\beta',\gamma')$.

Next, we investigate the behaviour of the implementation when performing *sequences* of internal transitions. We prove that $\mathsf{Impl}_{\alpha,\beta,\gamma}$ is strongly normalizing on τ-transitions: it may always reduce to $\mathsf{Impl}_{\widehat{\mathrm{T}}(\alpha,\beta,\gamma),0,0}$, and does so within a bounded number of τ-steps.

Lemma 4 (Normalization). *For all* $\mathsf{Impl}_{\alpha,\beta,\gamma}$*, we have that*

1. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\not\Rrightarrow}$ *iff* $\mathrm{spt}(\beta) = \emptyset = \mathrm{spt}(\gamma)$.
2. *there exists* $n \in \mathbb{N}$ *such that whenever* $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow}^k I$*, then* $k \leq n$.
3. *if* $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow}^* I \overset{\tau}{\not\Rrightarrow}$*, then* $I = \mathsf{Impl}_{\widehat{\mathrm{T}}(\alpha,\beta,\gamma),0,0}$.
4. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow}^* \mathsf{Impl}_{\widehat{\mathrm{T}}(\alpha,\beta,\gamma),0,0}$.

We now proceed to the main result of the paper, stating that the reachable states of the specification and of the implementation—in each case captured by the respective transition systems up to structural congruence—are precisely related.

Theorem 2. *The Binary Relation.*

$$\{\, (\,\mathsf{Spec}_{\widehat{\mathrm{T}}(\alpha,\beta,\gamma)},\, \mathsf{Impl}_{\alpha,\beta,\gamma}\,) \;\mid\; \mathsf{Impl}_{\alpha,\beta,\gamma} \textit{ is defined} \,\}$$

is a weak $\Rrightarrow$*-bisimulation.*

Corollary 3. $\mathsf{Spec} \approx \mathsf{Impl}$.

Proof. Since $\mathsf{Spec} \equiv \mathsf{Spec}_0$ and $\mathsf{Impl} \equiv \mathsf{Impl}_{0,0,0}$, this follows from Theorem 2 and Proposition 1.

This equivalence does not by itself guarantee that the implementation is free from live-locks since weak bisimulation, although properly reflecting branching in transition systems, is not sensitive to the presence of infinite τ-sequences. However, their absence was proven in Lemma 4(*2*).

References

[AG99] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.

[BH00] M. Berger and K. Honda. The Two-Phase Commitment Protocol in an Extended pi-Calculus. In L. Aceto and B. Victor, eds, *Proceedings of EXPRESS '00*, volume 39.1 of *ENTCS*. Elsevier Science Publishers, 2000.

[Gos91] A. Goscinski. *Distributed Operating Systems, The Logical Design.* Addison-Wesley, 1991.

[Ing94] A. Ingólfsdóttir. *Semantic Models for Communicating Processes with Value-Passing.* PhD thesis, University of Sussex, 1994. Available as Technical Report 8/94.

[LT98] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. Technical Report MIT/LCS/TM 373, MIT Press, Nov. 1998.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPW92] R. Milner, J. Parrow and D. Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, Sept. 1992.

[NFM03] U. Nestmann, R. Fuzzati and M. Merro. Modeling Consensus in a Process Calculus. In R. Amadio and D. Lugiez, eds, *Proceedings of CONCUR 2003*, volume 2761 of *LNCS*. Springer, Aug. 2003.

[OEBH03] 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.

[OGEA$^+$03] 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.

[RD01] 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.

[RFH$^+$01] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. A Scalable Content Addressable Network. In *SIGCOMM 2001, San Diego, CA*. ACM, 2001.

[SMK$^+$01] 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*. ACM, 2001.