ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects

Karl Palmskog Mads Dam Andreas Lundblad
School of Computer Science and Communication
KTH Royal Institute of Technology
Stockholm, Sweden
{palmskog,mfd,landreas}@kth.se

Ali Jafari
School of Computer Science
Reykjavik University
Reykjavik, Iceland
ali11@ru.is

We present a formalized, fully decentralized runtime semantics for a core subset of ABS, a language and framework for modelling distributed object-oriented systems. The semantics incorporates an abstract graph representation of a network infrastructure, with network endpoints represented as graph nodes, and links as arcs with buffers, corresponding to OSI layer 2 interconnects. The key problem we wish to address is how to allocate computational tasks to nodes so that certain performance objectives are met. To this end, we use the semantics as a foundation for performing network-adaptive task execution via object migration between nodes. Adaptability is analyzed in terms of three Quality of Service objectives: node load, arc load and message latency. We have implemented the key parts of our semantics in a simulator and evaluated how well objectives are achieved for some application-relevant choices of network topology, migration procedure and ABS program. The evaluation suggests that it is feasible in a decentralized setting to continually meet both the objective of a node-balanced task allocation and make headway towards minimizing communication, and thus arc load and message latency.

1 Introduction

An important problem, made more relevant by recent interest in cloud computing, is how to decouple computational processes from the underlying physical infrastructure on which they execute. One motivation for such decoupling is to free applications from handling resource allocation issues, which can instead be taken care of in a provably correct and transparent fashion using generic, application-independent mechanisms [7]. Potentially, tasks can then be performed at the physical machine most suited at the moment, continually meeting global system requirements such as utilization and power consumption, or task-local requirements such as a response time.

We consider the problem of runtime adaptation of tasks in the context of a core subset of ABS [12], a language for modelling distributed object-oriented systems developed in the EU FP7 HATS project. In ongoing work [7, 8] we are developing network-aware semantics for different fragments of ABS with some novel features. Specifically, we let objects execute on network nodes connected point-to-point using asynchronous message passing links, and show how location independent routing in such a setting can be used to support efficient, transparent, and robust (lock-free) object migration. Here, we examine how adaptation can be performed in such a model by a controller process running on each node.

To enable precise reasoning and experiments on adaptability, we define three central Quality of Services (QoS) objectives against which a solution for runtime adaptation in our context can be assessed: node load, arc load and message latency. We abstract from many practical, implementation-level concerns when interpreting these objectives in our setting. The load for a node is the number of active tasks running on it. The load for an arc is the number of messages traversing the arc. The latency for a message is the number of hops needed to reach its destination. We then restrict our consideration of adaptability to the problem of how and when to migrate objects to achieve the objectives as well as possible, given a specific network topology, ABS program, and node-local procedure for managing migrations. Using a simulator which implements the key parts of our semantics, we have investigated how well objectives are met for some application-relevant choices of network topologies, programs and migration procedures.

Section 2 and 3 describe the ABS language and our novel ABS-NET semantics for execution of ABS programs in a network. Section 4 describes our approach to runtime adaptation via object migration. Section 5 describes the simulator, our benchmark scenarios, and simulation results; Section 6 concludes.

2 ABS Background

ABS [11] is a language and framework for modelling distributed object-oriented systems, developed in the EU FP7 HATS project. Core ABS [12] is a language which contains the main features of ABS: a functional level for expressing data structures and side-effect free internal computations of objects, and an object level for expressing concurrent objects, and communication among such objects via method invocation. The object level defines syntax—reminiscent of Java's—for interfaces, classes, methods, object creation and method calls. The object level is accompanied by a type system and an operational semantics which preserves well-typing. One consequence of well-typing is that many runtime errors are ruled out for type-checked programs; when an object makes a call to a method *m* using an object identifier *o*, there always exists an object associated with *o*, which is an instance of a class which implements *m*. ABS uses placeholders in the form of futures for the result of method calls, allowing a caller to avoid blocking until the result value is actually required. In the variant of Core ABS we consider, a single object is the unit of concurrency, as in the variant of Albert et al. [1]. This means that objects at runtime can be viewed as actors, communicating between themselves only via asynchronous message passing.

An example of an ABS interface with implementing classes is given below. The `CastNode` interface defines a method `aggregate`, which, when called on an object, performs a convergecast operation in the object-reference binary tree rooted at that object. Specifically, this means that if an object implementing `CastNode` is a leaf in the tree (an instance of `LeafNode`), it simply returns a locally known integer, but if the object has child nodes in the tree (an instance of `BranchNode`), `aggregate` is called on both of those objects and the results are added to the local integer and returned. In this way, the `aggregate` method for the object *o* always returns the aggregate (sum) of all local values in the binary tree of objects rooted at *o*. The variables `fLeft` and `fRight` in the implementation of `aggregate` hold the placeholders (futures) for integers that result from the asynchronous method calls. The values are then retrieved through the **.get** operator, which can cause blocking until the method call has finished.

```
interface CastNode {                class LeafCastNode(Int val) implements CastNode {
    Int aggregate();                    Int aggregate() { return val; }
}                                   }

class BranchCastNode(Int val, CastNode left, CastNode right) implements CastNode {
   Int aggregate() {
      Fut<Int> fLeft = left!aggregate();
      Fut<Int> fRight = right!aggregate();
      Int aggregateLeft = fLeft.get;
      Int aggregateRight = fRight.get;
      return val + aggregateLeft + aggregateRight;
    }
}
```

3 Network Model and Semantics

To reason about object adaptability with respect to environmental conditions, we bring selected parts of the infrastructure of a distributed system into our model, namely, network endpoints and links. Endpoints and links are modelled as graph nodes and arcs with FIFO-ordered message queues, respectively. Conceptually, a node consists of an interpreter layer, where local objects reside, and a node controller, which acts as a mediator between the environment and node-local objects, as illustrated in Figure 1. The dashed arrow in the figure signifies that the identifier of object o_2 is known by object o_1, allowing o_1 to send method invocations to o_2. The structure is similar to that used in other programming-language oriented distributed system models, e.g., a proposed semantics for future Erlang [18]. Here, the node controller also contains logic for decision-making on adaptability. Seen abstractly, adaptability in this context becomes the problem of deciding when and where to migrate objects to achieve the QoS objectives—with the added constraint that all reallocations must be decided locally at each node.

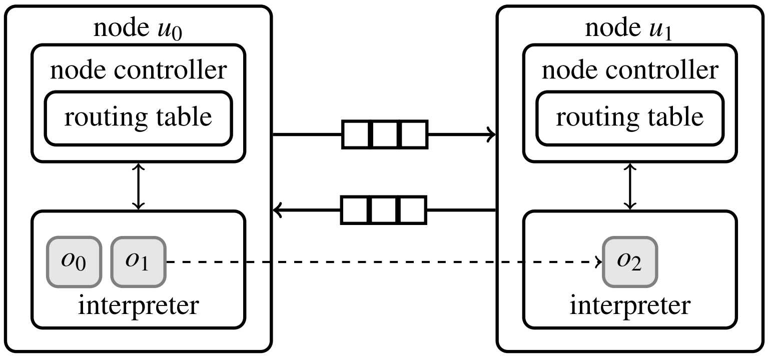

Figure 1: Nodes, node controllers, and interpreter layers.

To achieve location transparency, the basic problem is to route messages correctly between objects that have no prior, mutual knowledge of where they are located. Many solutions have been examined in the literature, including centralized or decentralized location servers, pointer chaining, and broadcast or multicast search. Sewell et al. [16] discuss many of these solutions, and their relative merits.

We are developing a novel approach to location transparency based on location-independent (also called name-independent) routing [7, 8], where the idea is to defer the maintenance of message routes to an explicit routing process executing independently of application-level messaging. Adapted to the approach suggested here, a node controller executing on each network node is responsible for maintaining routing information by exchanging routing tables with adjacent nodes in the network. This allows object migration to be supported in a transparent fashion with only modest extension to the runtime state.

We have defined a new operational semantics of Core ABS programs, in the same rewriting logic style [5] as the standard semantics, that characterizes task execution of objects located on, and moving between, network nodes. Adaptability features such as routing table exchange and object migration are modelled as nondeterministic events, with the node controller consisting of nothing more than a globally unique identifier and a routing table. We refer to the combination of the Core ABS functional layer, Core ABS object syntax, and our novel operational semantics as ABS-NET. We intend for the semantics to both guide implementation, by defining a baseline for retaining program runtime behaviour similar to Core ABS in a networked, decentralized setting, and provide opportunities for further theoretical analysis of specific adaptability strategies by refinement.

A complete description of the syntax and formal semantics of both Core ABS and ABS-NET is available in an appendix [14], with semantic equivalence explored elsewhere [8]. Below, we give an overview of the ABS-NET network model and semantics, with details on node controller behaviour, which is to an extent agnostic towards the underlying actor environment.

3.1 Runtime Configurations

The node controller's relationship with the interpreter layer residing on the node is symbiotic. On one hand, the node controller provides message delivery services and callback functions to obtain new globally unique object identifiers for objects residing in the interpreter layer. On the other hand, the node controller triggers object movement by using callback functions that the interpreter layer makes available. We assume a node controller is aware of the asynchronous links through which it can communicate via message passing with other node controllers. In essence, the aim is that node controllers should be realizable on top of a network with only OSI layer 2 interconnects, meaning that the required primitives for computation and communication can be implemented directly in hardware with high performance. If this is the case, running controllers on top of overlays using higher-layer interconnects such as sockets is also feasible, which is what we do in our simulator.

The global state in ABS-NET is formally a pair $\{net\}\{cn\}$. The network part $\{net\}$ is a set of nodes and arcs. In a node $\mathsf{nd}(u,\tau)$, u is a node identifier (assumed globally unique) and τ is a routing table, used to route object-related messages in the proper direction. In an arc $\mathsf{ar}(u,Q,u')$, representing a unidirectional link from u to u', Q is a FIFO-ordered queue of messages. The other part of the global state, $\{cn\}$, is a set of objects, with each object implicitly attached to a node in the network, and able to send and receive messages with the assistance of its host. A message *msg* can be (1) a table message $\textsc{Table}(\tau)$, used to pass a routing table τ from one node to another to update local routes, (2) an object message $\textsc{Object}(\mathit{object})$ containing a complete runtime object *object* to facilitate mobility, or (3) an application-level message transmitted from one object to another, which for ABS is either a method invocation ($\textsc{Call}$ message) or the resolved value of a future ($\textsc{Future}$ message). Given an application-level message, the function $\mathtt{dest}(\mathit{msg})$ returns the identifier of the intended recipient object, while the function `id` returns the identifier of a given runtime object.

The nature of a FIFO queue Q of messages is specified through three functions: `enqueue`, `dequeue` and `first`. $\mathtt{enqueue}(Q,\mathit{msg})$ returns the queue that results when the message *msg* is added to the back of Q. If Q is non-empty, $\mathtt{first}(Q)$ returns the message at the front of Q, and $\mathtt{dequeue}(Q)$ returns the queue that results when the front message is removed. For brevity, $\mathtt{enqueue}(Q,\mathit{msg}) = Q'$ is defined as a relation $Q \xrightarrow{\mathtt{enqueue}(\mathit{msg})} Q'$, while the conjunction that $\mathtt{first}(Q) = \mathit{msg}$ and $\mathtt{dequeue}(Q) = Q'$ is defined as $Q \xrightarrow{\mathtt{dequeue}(\mathit{msg})} Q'$.

The nature of a routing table is specified through the functions `update`, `next`, `register` and `replace`, and an infix operator $\in$. The function `update` takes three arguments: the routing table τ of the current node, the node identifier u' of the adjacent node, and the routing table τ' of the adjacent node. The function returns a routing table τ'', which incorporates the routes from τ' into τ if appropriate, with the constraint that all such routes must go through the node u'. For brevity, $\mathtt{update}(\tau,u',\tau') = \tau''$ is defined as a relation $\tau \xrightarrow{\mathtt{update}(\tau',u')} \tau''$. The function `next` takes three arguments: the routing table τ of the current node, the object identifier o' of the node we want the next hop for, and the default hop u, which is the identifier of the current node. The function returns the node identifier u' which is the next hop of o' according to the table. The function `register` takes four arguments: the routing table τ of the current node, the object identifier o' of the object we want to add a route for, the node identifier u of a neighbour node (usually self) which is the next hop, and a non-negative integer k for the distance to the object (in all instances in the rules, it is 0). The function returns a routing table τ' which incorporates the new route. For brevity, $\texttt{register}(\tau, o', u, k) = \tau'$ is defined as a relation $\tau \xrightarrow{\texttt{register}(o',u,k)} \tau'$. The function `replace` takes four arguments (of the same type as `register`): the routing table τ of the current node, the object identifier o' of the object we want to replace the route for, the node identifier u of a neighbour node which is the next hop, and a natural number k for the distance to the object. The function returns a routing table τ' which has removed any existing routes for o' and added the route given. For brevity, $\texttt{replace}(\tau, o', u, k) = \tau'$ is defined as a relation $\tau \xrightarrow{\texttt{replace}(o,u,k)} \tau'$. The claim $o \in \tau$, with a node identifier u given by the context, means that, according to τ, the object with identifier o is located on the node u.

The two parts of the global state can evolve jointly by performing synchronized labelled transitions, but also separately without exchanging information. The rules for such synchronization and separate evolution are shown in Figure 2. A label α is either $\texttt{mv}(object)$ (moving an object), $\texttt{rg}(o, o')$ (registering a new object identifier), or $\texttt{tr}(o, msg)$ (transporting a message). Intuitively, a label with an overline means that information is outgoing or being sent, while a label without overline means information is incoming or being received. An ABS-NET execution of an ABS program is a possibly infinite sequence of global states, such that the transition between a previous state and the next is valid. The program is not explicitly represented in a state, since it assumed to always be available unaltered at all nodes. The network topology is static during an execution, and we do not consider failures such as message loss.

$$\text{(NET-RED)}\ \frac{\{net\} \to \{net'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn\}} \qquad \text{(CN-RED)}\ \frac{\{cn\} \to \{cn'\}}{\{net\}\,\{cn\} \to \{net\}\,\{cn'\}}$$

$$\text{(CN-OUT-NET-IN-RED)}\ \frac{\{cn\} \xrightarrow{\overline{\alpha}} \{cn'\} \quad \{net\} \xrightarrow{\alpha} \{net'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn'\}} \qquad \text{(NET-OUT-CN-IN-RED)}\ \frac{\{net\} \xrightarrow{\overline{\alpha}} \{net'\} \quad \{cn\} \xrightarrow{\alpha} \{cn'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn'\}}$$

Figure 2: ABS-NET reduction rules connecting objects and networks.

Intuitively, transitions by the rule NET-RED are driven by node-related events, e.g., timeouts triggering routing table exchanges, or application-level messages being received and routed further. Transitions by CN-RED are triggered when objects execute ABS program statements that only change internal state. CN-OUT-NET-IN-RED is used when program execution requires interaction with the environment to proceed (e.g., method calls) and for object migration. NET-OUT-CN-IN-RED is used when a node transmits objects or application-level messages received through links to the local interpreter layer.

3.2 Node Controller Behaviour

The reduction relation for networks, alluded to Figure 2, is defined by the rules in Figure 3. The rules apply to subsets of nodes and arcs, such that the elements can be rearranged to match the left-hand side. The labelled-transition rules NET-MSG-RECV-OUT, NET-MSG-SEND-IN, NET-OBJECT-SEND-IN and NET-NEW-OBJECT-IN, where a node exchanges information with an object, all use the premise $o \in \tau$ to restrict actions to pertain to node-local objects. This is how object location is reflected in ABS-NET. $\texttt{fresh}(o)$ means that the identifier o is globally unique.

For proper progress in execution, we assume networks are such that (1) there are no dangling arcs referencing non-existent nodes, (2) for every arc between nodes there is an arc in the opposite direction, and (3) every node comes with a self-loop arc, i.e., an arc going from and to the node. Self-loop arcs are important for two reasons. First, it allows us to use the same rules for message passing in both the case where the sender object is at a different node from the receiver object, and where the sender is at the same node as the receiver. Once a message has been put in the self-loop queue, it appears as if it

$$\text{(NET-TABLE-SEND)}\quad \dfrac{u' \neq u \qquad Q \xrightarrow{\texttt{enqueue}\,(\textsc{Table}\,(\tau))} Q'}{\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,Q,u') \rightarrow \mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,Q',u)}$$

$$\text{(NET-TABLE-RECV)}\quad \dfrac{Q \xrightarrow{\texttt{dequeue}\,(\textsc{Table}\,(\tau'))} Q' \qquad \tau \xrightarrow{\texttt{update}\,(\tau',u')} \tau''}{\mathsf{ar}\,(u',Q,u)\,\mathsf{nd}\,(u,\tau) \rightarrow \mathsf{ar}\,(u',Q',u)\,\mathsf{nd}\,(u,\tau'')}$$

$$\text{(NET-MSG-RECV-OUT)}\quad \dfrac{Q \xrightarrow{\texttt{dequeue}\,(msg)} Q' \qquad \texttt{dest}\,(msg) = o \quad o \in \tau}{\mathsf{ar}\,(u',Q,u)\,\mathsf{nd}\,(u,\tau) \overset{\overline{\texttt{tr}\,(o,msg)}}{\longrightarrow} \mathsf{ar}\,(u',Q',u)\,\mathsf{nd}\,(u,\tau)}$$

$$\text{(NET-MSG-SEND-IN)}\quad \dfrac{o \in \tau \quad \texttt{dest}\,(msg) = o' \qquad \texttt{next}\,(\tau,o',u) = u' \qquad Q \xrightarrow{\texttt{enqueue}\,(msg)} Q'}{\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,Q,u') \overset{\texttt{tr}\,(o,msg)}{\longrightarrow} \mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,Q',u')}$$

$$\text{(NET-ROUTE-FURTHER)}\quad \dfrac{Q_1 \xrightarrow{\texttt{dequeue}\,(msg)} Q_1' \quad \texttt{dest}\,(msg) = o \quad o \notin \tau \qquad \texttt{next}\,(\tau,o,u) = u'' \quad Q_2 \xrightarrow{\texttt{enqueue}\,(msg)} Q_2'}{\mathsf{ar}\,(u',Q_1,u)\,\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,Q_2,u'') \rightarrow \mathsf{ar}\,(u',Q_1',u)\,\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,Q_2',u'')}$$

$$\text{(NET-OBJECT-SEND-IN)}\quad \dfrac{o \in \tau \quad u' \neq u \qquad \tau \xrightarrow{\texttt{replace}\,(o,u',1)} \tau' \qquad Q \xrightarrow{\texttt{enqueue}\,(\textsc{Object}\,(object))} Q'}{\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,Q,u') \overset{\texttt{mv}\,(object)}{\longrightarrow} \mathsf{nd}\,(u,\tau')\,\mathsf{ar}\,(u,Q',u')}$$

$$\text{(NET-OBJECT-RECV-OUT)}\quad \dfrac{\texttt{id}\,(object) = o \qquad Q \xrightarrow{\texttt{dequeue}\,(\textsc{Object}\,(object))} Q' \qquad \tau \xrightarrow{\texttt{replace}\,(o,u,0)} \tau'}{\mathsf{ar}\,(u',Q,u)\,\mathsf{nd}\,(u,\tau) \overset{\overline{\texttt{mv}\,(object)}}{\longrightarrow} \mathsf{ar}\,(u',Q',u)\,\mathsf{nd}\,(u,\tau')}$$

$$\text{(NET-NEW-OBJECT-IN)}\quad \dfrac{\texttt{fresh}(o') \quad o \in \tau \qquad \tau \xrightarrow{\texttt{register}\,(o',u,0)} \tau'}{\mathsf{nd}\,(u,\tau) \overset{\texttt{rg}\,(o,o')}{\longrightarrow} \mathsf{nd}\,(u,\tau')}$$

Figure 3: Node controller reduction rules.

came from some other node, and the rule NET-MSG-RECV-OUT can be applied. Second, it is not always the case that there is a route to the recipient of a message, because of the possibility of stale routing tables. However, messages must be dealt with somehow, in particular if they are coming from some other node, from which there could be other important messages pending. Hence, they are put in the self-loop queue, i.e., the default next hop of an application-level message is the node itself.

The ABS-NET reduction rules for objects at runtime are deferred to the appendix. Compared to Core ABS, the state of an object has been extended with an input and an output queues for asynchronous transfer of application-level messages, and a structure to keep track of resolved future values. In contrast, the standard Core ABS semantics handles resolution and querying of futures in a centralized way. In fact, all rule premises from the standard semantics that pertain to more than object-local state are absent in ABS-NET—its decentralized nature is syntactically apparent.

4 Adaptation

We consider three QoS objectives against which runtime adaptation solutions can be assessed: node load, arc load and message latency. In our setting, the definition of node load is simple but coarse grained: the load on a node u is the number of objects located on u with active tasks. One advantage of this measure is that it is an intrinsic property of runtime configurations. We need a model-intrinsic measure of load to enable reasoning at an abstract level about convergence to balanced allocation and that loads stay within a certain range. One disadvantage of the approach is that it fails to take into account the varying use of memory and processing power among tasks. However, in an implementation, a more fine-grained measure of load can be adopted, as long as it is linear in the number of active tasks.

We define the load of a particular arc as the number of messages traversing it per simulated unit of time. Hence, global minimization of arc load means that a minimal number of inter-node messages are sent overall, with respect to the current state of routing tables at nodes. Unless all routing tables are optimal (minimum stretch), however, there is no guarantee that the number of hops, i.e., latency, of a particular object-addressed message is minimal.

In our evaluation of runtime adaptation, we use ABS programs that are nonterminating and cyclical. The motivation is that for adaptations to current conditions to have a chance of conveying benefits, similar conditions must hold in the future. There is no obvious payoff in attempting to adapt when future states are random independently of the current state.

Although we wish to simultaneously meet all of our QoS objectives fully, we consider node load balancing our primary concern. Load balancing solutions are also relatively well-studied in the literature, making it easier to find a good starting point. Azar et al. [2] consider the problem of achieving balanced allocations in the framework of stochastic processes, where it is viewed as stepwise allocations of balls into bins. They highlight the use of greedy schemes for quickly converging to a ball-to-bin assignment where the maximum number of balls in any bin is minimized. The main drawback of this approach in a distributed setting is the reliance on atomic, single assignments of a ball to a bin at each algorithm step. Even-Dar and Mansour [9] study load balancing in a distributed setting where allocations are not necessarily done one-at-a-time. They give a distributed algorithm for selfish rerouting that quickly converges to a Nash equilibrium, which corresponds to a balanced resource allocation. However, at each round, locally computing a new allocation requires having exact knowledge of all loads in the system, which is complicated and costly to acquire in the current setting. Berenbrink et al. [3] describe and analyze fully distributed algorithms which require only local knowledge of the total number of resources and the load of one other resource to perform a single task migration step. The algorithms, some of which have attractive expected time for convergence, can be straightforwardly translated to a synchronous, round-based distributed setting and further to a message-passing setting, assuming some inherent synchrony. One important assumption made in the algorithm analysis is that a task can migrate to any other resource in a single concurrent round. For this property to hold, the underlying network graph must be complete, which we do not generally assume.

A factor in the convergence time is whether neutral moves are allowed, i.e., whether a migration can happen even when, as far as can be told locally, the move does not result in a more balanced allocation but merely an equally good one. For allocations in a sparse network graph where load differences between neighbours are one, there can nevertheless be maximal load differences in the order of the graph diameter, which can be significant. With neutral moves, such allocations can be improved on.

The problem of oscillating behaviour during task balancing can be mitigated by the use of coin flips before finalizing decisions to migrate tasks, as in the algorithms of Berenbrink et al. Oscillation can be worsened by reliance on stale information, but if the information is not *too* stale, oscillation periods can sometimes be bounded [10].

The literature on load balancing related to scientific computing contains work on simultaneously optimizing task allocations and communication overhead. For example, Cosenza et al. [6] give a distributed load balancing scheme for simulations involving agents moving in space from worker to worker. The scheme, which is validated experimentally, optimizes both worker load and communication overhead between workers, but assumes only a small area of interest for each agent, with agents unable to communicate with other agents outside this area. In the current work, two objects can communicate whenever the identifier of one of them is known to the other, making it harder to minimize communication overhead. Catalyurek et al. [4] describe how to use hypergraph partitioning to minimize both communication volume and migration time of tasks for parallel scientific computations. However, the repartitioning is performed in batch and requires complete knowledge of the data and computations on each node.

At this initial stage of the work, we do not consider the cost of migration itself in terms of messaging and other resources. Hence, we only measure communication in terms of messages exchanged between objects, ignoring overhead in terms of routing and load-related messages.

5 Simulator

We have evaluated our runtime adaptation approach by developing a simulator for running ABS programs in a network of nodes according to the ABS-NET semantics. We have run the simulator with a variety of network node topologies and object migration procedures on a number of proposed synthetic scenarios defined by ABS programs.

Our simulator's main purposes are to serve as a proof-of-concept for ABS-NET and to allow us to run adaptability case studies with particular programs and topologies. Specifically, we are interested in studying convergence properties of object migration procedures in practice, and in showing that our approach to distributed execution scales to networks with many nodes.

The simulator is implemented in Java. Each node controller is implemented as a Java thread, which communicates with other controllers through TCP sockets, using the KryoNet network library [13]. One reason for the choice of sockets is to enable to scale simulations over several physical machines and a large number of simulated network nodes. All node controllers in the network have a representation of the abstract syntax tree of the ABS program being executed, which is generated from ABS program code by the lexing and parsing frontend shared by most ABS backends.

As in the conceptual model and the formal semantics, a node controller can have zero or more objects, each having at most one active task. An active task has a reference to the statement currently being executed in the abstract syntax tree. We call an object active if it has an active task. Scheduling of active tasks is done at the node controller level in a round-robin fashion for active objects. More precisely, the scheduler deterministically steps all active tasks, checks for active objects, and then repeats the process on the new set of active tasks.

We implement statement execution by interpretation. The main reason for this choice is to enable easy serialization of objects between executing statements; to get immediate results from load balancing, we must be able to migrate active objects. One drawback of using interpretation is that local execution is slow and resource-demanding compared to execution in the standard ABS backends.

A node controller is associated with a unique TCP port on the host system. Besides a list of neighbour handles, which abstract over underlying sockets, and a list of local objects, the node controller maintains a routing table. The routing table is broadcasted to neighbours after entries have been changed or added as a result of statement execution or incorporation of routes from neighbour messages. Hence, except when many locations have been updated in a short interval, we expect routing tables to be up-to-date or nearly so, taking into account the network size restrictions of the simulator.

Network topology setup and program loading is handled by scripting on top of a custom simple command-line interface (CLI). When starting up, a node controller is assigned a migration procedure through the CLI, which is the same for all node controllers in the network. One desirable feature that is not implemented is CLI control of link characteristics, such as delays.

By default, the simulator starts the initial task on a single startup node. The initial task is defined by the statements in the mandatory starting block of the ABS program. In all our programs, this task creates all the objects used for the duration of the program. Migration and logging does not commence until a method with the name `setupFinished` is called on some object. There are several reasons for this kind of initialization: it is easier to predict load balancing behaviour with a fixed set of objects, and it is problematic to create new objects on the fly without garbage collection, which we have not implemented.

5.1 Scenarios

A network configuration determines the size and topology of the network; large and dense networks give more overhead in the form of routing and load messaging, making simulations slower. Currently, highly connected topologies with in the order of 25 network nodes can be simulated in reasonable time. On this note, we limit the evaluation to networks with three distinct underlying network topologies from sparsely to fully connected: grids, hypergraphs and full meshes. Our base initial setup for each topology has 32 nodes. Since the simulator scales to at least in the order of 100 nodes for sparsely connected topologies, we also investigate grids larger than 32 nodes to compare results.

For defining object behaviour, we have developed a number of ABS programs specifically to run in our simulator. All programs have a setup phase, where a fixed number of objects are initialized, and a phase where the generated objects perform some computation, possibly involving communication; there are no short-lived dynamically created objects. For all programs but one, which implements the Chord distributed hash table (DHT) algorithm [17], communication patterns among generated objects follow straightforwardly from the code. This makes it easier to follow what happens during a simulation and to reason about how far an allocation of objects to nodes is from the optimum, factors which we considered particularly important in scenario development. After running initial simulations, we have adjusted parameters in our programs, and in some cases added functionally redundant instructions to get constant and consistent load and messaging, since our migration procedures consider mainly objects with active tasks. With spurious activity among nodes, messaging and load varies greatly, and progress becomes hard to discern. The programs below are available online [15]:

`IndependentTasks.abs` The starting task generates objects, and each generated object is called upon to perform a long-running task. There is no communication among workers—only briefly at startup between the coordinator object, which initializes and assigns tasks, and the generated objects. Since there is no communication, an optimal allocation is an even distribution of objects among nodes, regardless of the network topology.

`Star.abs` An object star configuration consists of one center object and one or more fringe objects. The fringe objects in the star continually communicate with the center object, but not among themselves. The program builds a number of independent object star configurations.

`Ring.abs` The starting task generates objects which know the identifiers of the next object in the ring. The last object generated gets the identifier of the first object. The first object, when called, calls its next object, and so on, until the object which has the first object as next object is reached. In the computation phase, many such calls traverse the ring simultaneously.

`ChordDHT.abs` An implementation of the Chord DHT algorithm. Key-value mappings are distributed between a number of objects, which all support a put/get interface to clients. Objects are arranged in a ring, but aside from references to their neighbours, each object has $\log(n)$ "fingers", references to non-adjacent objects, where n is the size of the keyspace. The addition, or join, of an object to the ring places the new object at a particular position based on its identifier and can trigger global reconfiguration of the ring. During setup, 128 objects are joined to the chord, and each object becomes associated with either a producer object, which continually puts values into the DHT, or a consumer object, which continually attempts to retrieve values from the DHT using pseudorandom keys.

We consider only migration procedures that as a first priority balance out load evenly among nodes in the network. As a consequence, a simulated node controller continually informs neighbour nodes of its load when appropriate, and receives load messages from neighbours in turn, regardless of the migration procedure used. In the simulator, each migration procedure defines a callback method which takes the affected node controller as a parameter. The callback method is invoked, and can result in the migration of several objects to neighbour nodes. The migration procedures used are described below.

Berenbrink et al. An adapted version of the distributed load balancing algorithm by Berenbrink et al. [3], which does not allow neutral moves. One notable difference in the simulator implementation from the abstract description given in Algorithm 1 is that only a fixed small number of objects (20) have the possibility to migrate in each cycle, because of limits on the sizes of message buffers.

Berenbrink et al. with neutral moves An adapted version of the distributed load balancing algorithm by Berenbrink et al., which does allow neutral moves, and therefore converges more slowly. The only difference from Algorithm 1 is that the if-condition is $l > l'$ instead of $l > l' + 1$. As determined experimentally, only migrating one or two objects per node per cycle leads to significantly less oscillation of objects, compared to when migrating three or more.

Berenbrink et al. with communication intensity A variant of the preceding procedure, where objects are selected for migration based on their affinity to the (randomly) chosen neighbour node, as determined by their communication history with objects in the neighbour node's direction. The communication history is a list of other objects that a given object has communicated with recently, as given by abstract object-local time, defined by the number of tasks finished since initialization. The affinity of an object to the neighbour node is then quantified as the number of objects in the communication history that are located in the direction of the node, according to the routing table.

Weighted neighbour load difference Once every cycle, an object and an adjacent node are chosen uniformly at random and independently. Then, a probability of migration is calculated and enacted based on the difference in load between the current node and the chosen node, with probability 1 for a difference of 10 or more, and probability 0 for a negative difference. If the load difference is d, the migration probability becomes $\frac{d}{10}$, adjusted to closest number in the interval $[0, 1]$.

Weighted neighbour load difference with communication intensity Given a randomly chosen object and adjacent node as in the previous procedure, we define the probability of migration according to communication intensity as the number of entries in the object's communication history found in the direction of the node, divided by the total number of entries in the history. This probability is then combined via weighted averaging with the neighbour load difference probability to define the weighted neighbour load with communication procedure. We have used the weight 0.2 for the communication intensity probability and 0.8 for the neighbour's load probability.

Algorithm 1 Berenbrink et al. load balancing cycle.

```
for each active object $o$ do
    let $u'$ be a neighbour chosen uniformly at random
    let $l$ be the current load, let $l'$ be the last known load of $u'$
    if $l > l' + 1$ then send $o$ to $u'$ with probability $1 - l'/l$
```

5.2 Scenario Objectives

Since our primary objective is to balance node load evenly, we record the load of all individual nodes over time, and then compute the maximum load and load standard deviation. For scenarios with little to no object communication, these are the only measures that are relevant with respect to our objectives. For scenarios with significant messaging, we also consider the number of object-related messages sent (i.e., CALL and FUTURE messages) by each node between sampling intervals—with the average number of messages and standard deviation shown. We do not count messages sent by a node to itself via the self-loop arc, since such messages need not go through a physical link in an implementation.

We sample the required quantities from simulations at a fixed global rate, corresponding roughly to a certain number of transitions (1000) in the semantics with imposed fairness via round-robin scheduling.

5.3 Results

In this section, we describe simulation results for the scenarios given above.

5.3.1 Simulations of `IndependentTasks.abs`

The program creates 201 objects in total: one starting object which becomes inactive after initialization and 200 objects that each have a task that runs for the course of the program.

As expected, the algorithm by Berenbrink et al. without neutral moves converges very quickly and stays unchanged with no migrations after reaching a state where neighbour load differences are at most one, which on a full mesh is always balanced. For most of the runs on a 32-node hypergraph network topology, the stable state coincided with a completely balanced allocation, or very closely so. For the grid case, the stable allocation in almost all cases deviated significantly from a fully balanced one.

The algorithm variant with neutral moves and two migrations per cycle converges to an almost-stable state quite quickly on a hypergraph, but continues to have minor oscillation of objects. With the same algorithm and five object migrations allowed per cycle, there is considerably more oscillation going on after coming close to a balanced allocation. On a grid topology, where a stable allocation can be further away from a balanced allocation, allowing neutral moves gives better results than disallowing them, as expected. For a grid, the gain from using neutral moves is most distinct in a lower standard deviation compared to the algorithm without neutral moves.

5.3.2 Simulations of `Star.abs`

In the star program, stars are constructed so that each node can hold a whole star, and there is precisely one star per network node. In an optimal allocation, therefore, there are no node-to-node message exchanges at all; all messages are sent locally.

We expected the pure load balancing procedures to have markedly worse results than the procedures taking inter-object communication intensity into account. The average number of sent messages and the standard deviation of sent messages over time for the star program on a grid is shown in the upper half of Figure 4, with measurements smoothed out via averaging over five samples to reduce noise. As can be seen, there is a distinct improvement with respect to messages sent when using the algorithm by Berenbrink et al. augmented with message intensity comparisons when compared to the other procedures, although it is quite far from the optimum. The algorithm using probabilistic weighting of load and messaging seems to improve the most over time, although it performs similarly to the messaging-augmented load balancing algorithm by Berenbrink et al.

With all the tested migration strategies for a grid, load became evenly balanced relatively quickly, as seen in the lower left part of Figure 4. Hence, there was no significant avoidance of messaging by communicating objects clustering at a few specific nodes.

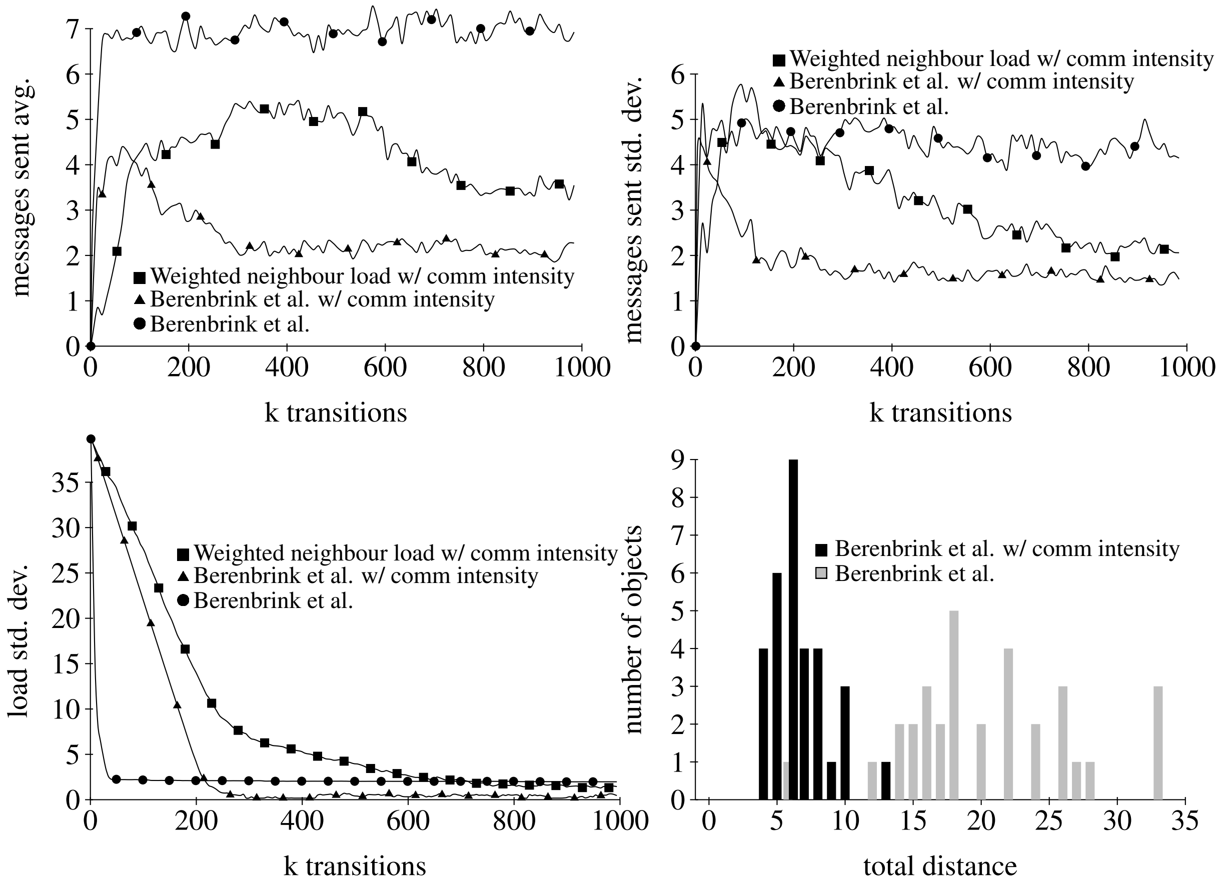

Figure 4: `Star.abs` running on a 32-node grid.

Because of the simplicity of the object communication graph and the fact that it is possible to reach an allocation where no inter-node communication takes place, it is worthwhile to illustrate how near specific algorithms can get after many (1000) cycles, for comparison. In a given allocation, each object has a total distance in hops to the other object it communicates with. For fringe objects, the total distance is the number of hops to its center object, but center objects have total distance equal to the sum of all distances to its fringes. In an optimal allocation, all centers (and all fringes) have total distance zero. In the lower right part of Figure 4, gray bars show the distribution of total distance among the 32 center objects on a grid for the load balancing algorithm by Berenbrink et al. The black bars show the distribution of total distances of the objects for the algorithm by Berenbrink et al. augmented with message intensity comparisons. The distributions intersect, but the former algorithm fares worse.

Results for `Star.abs` on a hypergraph topology give a less pronounced advantage to the two migration procedures which take message intensity into account. Of those procedures, the Berenbrink et al. variant produces the least messaging, but trends are largely the same as for the grid case; hence, we omit plots. For the case of a complete topology, the amount of messaging was virtually the same for all procedures. An intuition for why this is the case is that it becomes much harder to improve upon an allocation in a situation where migrations are helpful only when communicating objects end up on the same node, and there is additionally no corresponding loss of proximity to another object. Grids of 64 and 128 nodes have the same messaging trends as the 32-node case.

5.3.3 Simulations of `Ring.abs`

When running a ring of 128 objects on a 32-node grid, there are balanced allocations with all nodes having 4 objects, where all objects that communicate are either on the same node or adjacent nodes. The idea is that two of the objects on a node are part of a segment of the ring, while the other two are part of another segment coming back the other way. In such allocations, at most one inter-node message per object is needed for a method invocation that involves the whole ring.

The upper left half of Figure 5 shows the average number of messages sent of a 128-object ring on a grid topology, while the upper right half shows the standard deviation of the number of sent messages; smoothing by averaging samples has been applied in both cases. The pattern from the star program remains, with procedures taking messaging into account performing better, but the differences are smaller. The progressively decreasing number of inter-node messages sent are not due to clustering of many objects on a few nodes, as shown by the eventually low standard deviation of load in the lower left part of the figure.

In the lower right part of Figure 5, gray bars show the distribution of total distance among all ring objects on a grid to the objects they communicate with, after 1000 migration cycles using the algorithm by Berenbrink et al. Black bars show the distribution for the algorithm by Berenbrink et al. with neutral moves augmented with message intensity comparisons. There is overlap, but the latter algorithm results in many more objects with total distance between 1 and 5. However, both distributions are quite far from being optimal.

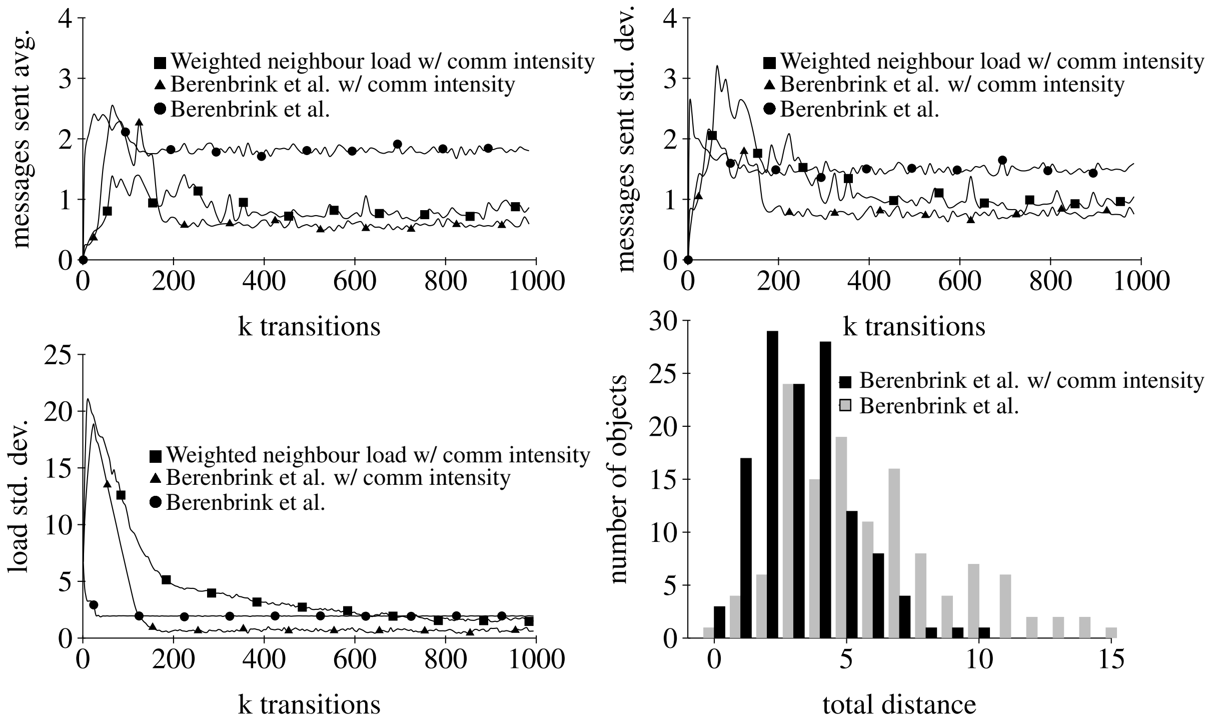

Figure 5: `Ring.abs` on grid.

As in the case of `Star.abs`, the performance trend in messaging over time is largely the same on a grid and hypergraph topology for `Ring.abs`. The main difference on a hypergraph is that procedures which take message intensity into account result in less pronounced improvements over the pure load balancing procedure. For a complete topology, differences are once again small, but with an edge towards the message intensity procedures. Once more, grids of 64 and 128 nodes preserve the trend from the 32-node case.

5.3.4 Simulations of `ChordDHT.abs`

In the Chord DHT program, the weighted neighbour's load and message intensity strategy exhibited a tendency to quickly cause message buffer overflows, while the procedures based on the algorithms of Berenbrink et al. worked largely as expected.

The left part of Figure 6 shows the average number of messages sent for nodes when running the program on a grid, and the right part shows the standard deviation of the number of messages. Again, smoothing by averaging samples five at a time has been applied. The results suggest that there is a reasonable payoff from taking messaging into account in a migration strategy, even when running a program with relatively complex communication patterns.

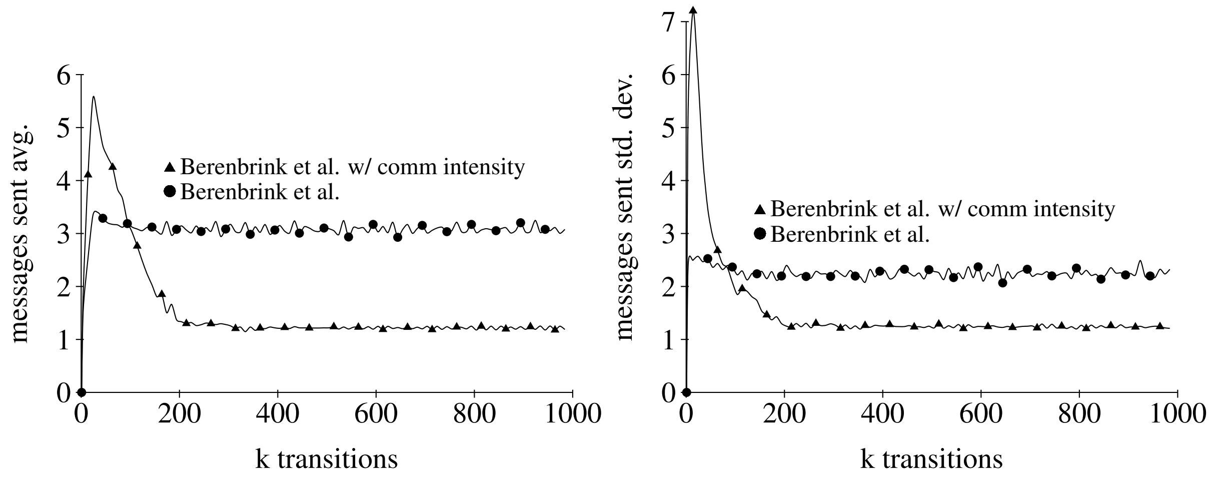

Figure 6: `ChordDHT.abs` on grid.

Simulations of `ChordDHT.abs` on a hypergraph show very similar trends in performance to the grid case, but give a less pronounced advantage to procedures which take message intensity into account, as for previous programs. In a fully connected topology, the procedures result in effectively the same amount of messaging, as before.

6 Conclusions and Future Work

The simulation results suggest that it is feasible in a decentralized setting to meet the objective of balanced resource allocation, and also make headway towards the objective of minimizing communication of distributed objects. The results also validate the applicability of the ABS-NET model with location-independent routing to decentralized runtime adaptation. The main concern for relevance to real-world networks is the use in the model of unbounded message queues, and the lack of rate limitation and latency controls in our simulator.

In future work, we plan to continue the theoretical and simulation-based studies to deepen our understanding of multi-dimensional resource management, to improve the performance and accuracy of the simulator, and to investigate adaptation in dynamic networks, initially only with benign churn, i.e., with controlled startup and shutdown of nodes.

References

[1] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte & S. L. Tapia Tarifa (2011): *Simulating concurrent behaviors with worst-case cost bounds*. In: *Proceedings of the 17th international conference on Formal methods*, FM'11, Springer-Verlag, Berlin, Heidelberg, pp. 353–368.

[2] Y. Azar, A. Z. Broder, A. R. Karlin & E. Upfal (1994): *Balanced Allocations*. In: *SIAM Journal on Computing*, pp. 593–602.

[3] P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. W. Goldberg, Z. Hu & R. Martin (2006): *Distributed selfish load balancing*. In: *Proc. 17th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pp. 354–363.

[4] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy & L. A. Riesen (2007): *Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations*. In: *Proc. of 21st IEEE International Parallel & Distributed Processing Symposium*.

[5] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer & C. Talcott (2007): *All About Maude - A High-Performance Logical Framework*. Springer.

[6] B. Cosenza, G. Cordasco, R. De Chiara & V. Scarano (2011): *Distributed Load Balancing for Parallel Agent-Based Simulations*. In: *Proc. 19th Conf. on Parallel, Distributed and Network-Based Processing*, pp. 62 –69.

[7] M. Dam: *Location Independent Routing in Process Network Overlays*. Manuscript, to be submitted, `http://www.csc.kth.se/~mfd/Papers/lirpno.pdf`.

[8] M. Dam & K. Palmskog: *Efficient and Fully Abstract Routing of Futures in Object Network Overlays*. Manuscript, to be submitted, `http://www.csc.kth.se/~mfd/Papers/efarfono.pdf`.

[9] E. Even-Dar & Y. Mansour (2005): *Fast convergence of selfish rerouting*. In: *Proc. 16th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pp. 772–781.

[10] S. Fischer & B. Vöcking (2005): *Adaptive routing with stale information*. In: *In Proc. 24th Ann. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC)*, ACM, pp. 276–283.

[11] FP7-231620 (HATS) Project (2011): *Deliverable 1.2: Full ABS Modeling Framework*. `http://www.hats-project.eu`.

[12] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte & M. Steffen (2010): *ABS: A Core Language for Abstract Behavioral Specification*. In: *Proc. of Software Technologies Concertation on Formal Methods for Components and Objects*.

[13] KryoNet authors: *KryoNet project*. `http://code.google.com/p/kryonet/`.

[14] K. Palmskog: *The Formal Semantics of Core ABS and ABS-NET*. `http://www.csc.kth.se/~palmskog/abs-net/semantics-abs-net.pdf`.

[15] K. Palmskog, M. Dam, A. Lundblad & A. Jafari: *ABS-NET definitions and programs*. `http://www.csc.kth.se/~palmskog/abs-net/`.

[16] P. Sewell, P. T. Wojciechowski & A. Unyapoth (2010): *Nomadic Pict: Programming Languages, Communication Infrastructure Overlays, and Semantics for Mobile Computation*. *ACM Transactions on Programming Languages and Systems* 34.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek & H. Balakrishnan (2001): *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. In: *Proc. ACM SIGCOMM '01*.

[18] H. Svensson, L.-Å. Fredlund & C. Benac Earle (2010): *A Unified Semantics for Future Erlang*. In: *Proc. of Erlang '01*.