ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects

February 28, 2013

Abstract

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 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 decoupling processes in a distributed infrastructure is that it becomes possible handle resource allocation at layers lower than the application layer. Potentially, tasks can then be performed at the physical machine most suited at the moment, continually meeting global system requirements for e.g. even utilization and task-local requirements such as a bounded response time.

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 handled in a transparent fashion using generic, application-independent mechanisms. Potentially, tasks can then be performed at the physical machine most suited at the moment, continually meeting global system requirements for, e.g., utilization, 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 [14], a language and framework for modelling distributed object-oriented systems developed in the EU FP7 HATS project.

Following our preceding work [8, 9], we construct a networked structural operational semantics for the ABS subset in rewriting logic style [6], where objects execute on network nodes connected point-to-point using asynchronous message passing links. We showed previously how object migration could be supported in an efficient, transparent, and robust (lock-free) manner using location independent routing. In the present work, we examine how adaptation can be performed in the networked 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 which a solution for runtime adaptation in our context can be assessed against: 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 specific node at a specific time is simply the number of active tasks running on it. The load for a specific arc is the number of messages traversing the arc. The latency for a specific 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 fulfilled for some application-relevant choices of network topologies, programs and migration procedures.

One potential application of our work is as a basis for a decentralized middleware system with very few dependencies and assumptions running on a networked ("cloud") infrastructure, that allows the provider to get high resource utilization when running resource-oblivious programs from a third party.

1.1 Contributions

We show that extending a general and reasonably practical object-oriented language (ABS) to execute in a network is feasible, and highlight the issues involved, from extending the semantics to concerns at implementation. The techniques apply to similar languages. Given that a language has a formal semantics, the extension process can be carried out formally. We also investigate parts of the design space for distributed adaptability algorithms in our decentralized setting.

2 ABS Background

ABS [13] is a language and framework for modelling distributed object-oriented systems, developed in the FP7 HATS project. In contrast to design-oriented languages such as UML, ABS offers constructs for expressing concurrency and the possibility to execute models according to an operational semantics descended from Creol [15]. In contrast to foundational concurrency-oriented languages such as the π-calculus [17], ABS provides higher-level primitives that can be used to directly model object-oriented systems.

Core ABS [14] is a language which contains the main features of ABS: a functional level for expressing data structures and side-effect free internal computations of distributed objects, and an object level for expressing concurrent objects, and communication among such objects via method invocation. The object level defines syntax for interfaces, classes, methods, object creation and method calls, where there is inheritance among interfaces but not among classes. The object level is accompanied by a type system and a structural operational semantics which preserves well-typing. Hence, method invocations in Core ABS cannot go wrong at runtime 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 an interface where m is defined.

The runtime unit of concurrency in Core ABS is a concurrent object group (cog). A cog contains one or more runtime objects, which perform cooperative scheduling of tasks. We use a variant of Core ABS where a single object is the unit of concurrency rather than a cog, similar to the variant of Albert et al. [1]. The choice is motivated by our focus on network adaptability of individual objects and computation tasks, which becomes more complicated when objects in a group must perform intermittent synchronization. In this language variant, all individual objects can be interpreted as actors, having local store and communicating with the environment only via asynchronous message passing. Additionally, our language variant fixes a number of minor inconsistencies in the syntax and semantics of the original Core ABS, for example by prohibiting multiple return statements which could cause unexpected nonterminating behaviour. The language is described in detail at the accompanying website (`http://www.csc.kth.se/~palmskog/abs-net/`).

A fragment of a Core ABS program is given as an example in Figure 1. The `CastNode` interface defines a method `aggregate`, which, when called on some object, is intended to perform a convergecast operation in the binary tree rooted at that object. Specifically, this means that if an object implementing `CastNode` is a leaf in the tree (an instance of class `LeafNode`), it simply returns a locally known integer, but if the object has child nodes in the tree (an instance of class `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 of all local values in the binary tree of objects rooted at o.

The implementation of the `aggregate` method in the program highlights the use in Core ABS of *futures* as placeholders for results from asynchronous method calls. The variables `fLeft` and `fRight` hold futures which ultimately resolve to integer values, as indicated by their type declarations. In the right hand side of the declarations of the futures, the delimiter '!' between the object variable name and the method name signifies asynchronous invocation, which always immediately returns a future. The usual dot delimiter '.' signifies a synchronous invocation which blocks the caller until the final result is returned without any intermediary.

Before returning the aggregate of the current object, the aggregate of each child node is retrieved by appending `.get` to the variable holding the respective future. Evaluations of assignments with this construct can be blocking, unless an `await` statement was executed first with the future variable involved, e.g. `await fLeft?;`. Executing `await` when the associated future has not yet been resolved does not force the caller into busy waiting; if there are method invocations for the object waiting be processed, control can be changed to the corresponding process at the discretion of the scheduler, and pass back to the original invocation later.

Informally, a Core ABS runtime configuration is a bag of objects and futures equipped with unique identifiers, along with unprocessed method invocations. An object in the bag has values for all variables defined in its class, a queue of processes representing received method invocations, and possibly an active process. Futures either have the value to which they resolve, or a placeholder to indicate that no resolution is available. When an asynchronous method invocation statement is executed, a method invocation is added to the bag, ready to be consumed by the callee. In contrast with actor languages such as Erlang and Rebeca [19], which provide the traditional guarantee that messages from one actor to another are always processed in the order they are sent, the Core ABS semantics does not prescribe any particular order for processing method invocations. In effect, the runtime environment provides an unbounded number of one-place buffers that objects can use to communicate with objects for which identifiers are known.

While interface names are proper type names in Core ABS, class names are not, and are thus only used in object creation with the `new` keyword. For example, the assignment `CastNode nd = new LeafCastNode(0);` creates a

```
interface CastNode {
    Int aggregate();
}

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

Figure 1: Core ABS example interface and classes.

`LeafCastNode` object with the `val` variable set to 0.

3 Network Model and Semantics

To reason about object adaptability 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, we consider a node to consist of an object 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 2. This node controller is not treated explicitly in the immediately related work [8, 9]. The dashed arrow in the figure signifies that an object identifier is known by another object and thus can be used for method invocation. The node controller also contains logic for decision-making on adaptability. Seen abstractly, adaptability here 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.

In order to support 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,

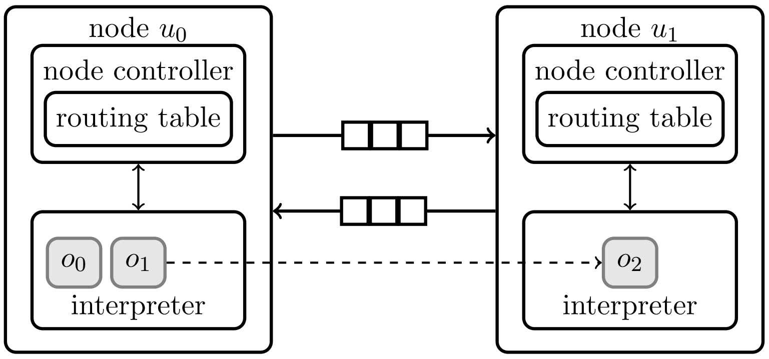

Figure 2: Nodes, node controllers, and interpreter layers.

including centralized or decentralized location servers, pointer chaining, and broadcast or multicast search. Sewell et al. [18] has an extensive discussion of the various approaches in the literature, and their relative merits.

We have previusly proposed a novel approach to location transparency based on location independent (aka name independent) routing [8, 9], 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 neighbouring nodes in the network. This allows object migration to be supported in a transparent fashion with only modest extension to the runtime state.

3.1 Operational Semantics

We have defined a networked semantics for Core ABS based on our previous work [8, 9]. Adaptability features such as routing table exchange and object migration are modeled 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 the associated structural operational semantics described below as ABS-NET. We intend for the semantics to both guide implementation, by defining a baseline for retaining program runtime behavioural similar to Core ABS in a distributed setting, and provide opportunities for further theoretical analysis of specific adaptability strategies by refinement.

3.1.1 ABS-NET Runtime Configurations

In our semantics, an ABS-NET runtime configuration consists of two disjoint subconfigurations: one bag *net* representing the network with nodes and arcs, and one bag *cn* of all objects located at nodes in the network. The fact that a particular object is located on a particular node is not explicitly represented, but reflected behaviourally in the reduction rules.

The bag *net* consists of nodes $\mathsf{nd}\,(u, \tau)$, where u is an identifier assumed to be globally unique, and τ is a routing table, and arcs $\mathsf{ar}\,(u, Q, u')$, where Q is an unbounded queue which buffers messages from node u to node u'.

The bag *cn* consists of objects $\mathsf{ob}\,(o, a, p, q, Q_{in}, Q_{out}, \Sigma)$, where o is an identifier assumed to be globally unique, a is a store for values of instance variables, p is the active process (if any), and q is a bag of inactive processes. Q_{in} is the message input queue (mailbox) and Q_{out} is the message output queue; both queues are unbounded. We do not represent futures directly; instead, each object is equipped with a structure Σ that contains a map from future identifiers to values. When a future value is needed by an object, it must be put into the map, which happens after a certain message is delivered by the node controller to the object. The motivation for an implicit future representation is mainly to keep the semantics uniform and of manageable size; first-class futures would require extending routing and mobility beyond objects. In an implementation, globally unique object identifiers can be achieved during object creation by adding a serial number to the identifier of the node the new object will be located at.

Network and object behaviour is defined in rewriting logic, in the same style as for Core ABS. All reduction rules are implicitly specified modulo commutativity and associativity of configuration composition, with the empty configuration ϵ as unit. Rules either define how a whole configuration changes, or how a matching subconfiguration changes while the remaining part stays unchanged. The former case is distinguished by the use of brackets around the configuration in the rule, e.g. $\{cn\}$. The following is an example of a pair of configurations describing a two-node network with a single object located on one of the nodes:

$$\{\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u, Q, u)\,\mathsf{ar}\,(u, Q', u')\,\mathsf{nd}\,(u',\tau')\,\mathsf{ar}\,(u', Q'', u')\}\ \{\mathsf{ob}\,(o, a, p, q, Q_{in}, Q_{out}, \Sigma)\}$$

The key rules for node controller and object behaviour are described below in Section 3.2 and Section 3.3, respectively. The complete set of language definitions and rules for both the Core ABS variant and ABS-NET are available on the accompanying website (`http://www.csc.kth.se/~palmskog/abs-net/`).

3.1.2 Handling Future Resolution

When representing futures implicitly as mappings stored in objects, there are at least two distinct strategies for how to ensure that future values are delivered through the network to the objects that need them, echoing the eager and lazy evaluation strategies in functional programming. In both strategies, the object which receives a message with a method call has an obligation to send back a message with the future value to the caller. The main difference between the strategies concerns what to do when an object shares a future identifier with another object by sending it as a parameter in a method call, as in the following Core ABS fragment:

```
Fut<Int> fArg = obj1!getArgument();
Fut<Int> fResult = obj2!getResult(fArg);
Int result = fResult.get;
```

The eager strategy assumes that all objects which receive a future identifier will attempt to retrieve the associated value; hence, whenever futures are transmitted, some action is taken which ultimately results in the resolved values of those futures being sent to the callee. In a lazy strategy, no particular future-related action is taken by the caller or any other entity. Instead, an object that actually needs a particular future value at run time requests it by sending special message, which is routed to the original object which originated the future identifier. The originating object then responds with a message containing the future value, once it becomes available.

Two drawbacks of an eager strategy are (1) that method call parameters must be inspected for occurrences of future identifiers, even when they are deeply nested in data structures, and (2) that messages containing future values are sent unnecessarily when callees simply ignore future identifier arguments. The lazy strategy avoids both of these drawbacks, but has drawbacks of its own. For example, retrieval of a future value, which can be costly and time-consuming, cannot happen concurrently with the processing that occurs before the future is needed. This means that if there is a sequence of blocking `.get` operations without `await`, all actions to retrieve the values will happen serially.

We have chosen to use an eager strategy for future resolution in our semantics. Specifically, each object maintains a list of futures for which it is obligated to forward resolved future values, and where those resolutions should be sent. Whenever a future value is shared with another object, e.g. through the arguments of a method invocation, the list is updated accordingly. When the object retrieves a resolved future value, the value is saved and forwarding by message passing according to the list becomes possible. Because futures can resolve to other futures, the forwarding list may need to be updated when a new future value is added. One reason for selecting this approach is that it is reasonably straightforward to pinpoint where (i.e., in which reduction steps) the future list, or map, needs to be updated, and what data needs to be considered.

3.1.3 Adaptability Assumptions

In the semantics, the Core ABS program being executed is assumed to be available unaltered at all nodes. The program is therefore not explicitly represented in the runtime configuration. Initially, we consider only networks that remain static over the course of program execution. We discuss briefly the case of benign dynamic networks, which is a planned extension, but leave all details of crash failures and byzantine failures for future work. On the same note, we also assume that messages sent between adjacent nodes cannot be lost—only ignored indefinitely as far as fairness permits.

We also assume that the behaviour of the program running on network nodes is nonterminating and cyclical. This assumption is motivated by our focus on adaptability; for adaptations to current conditions to have a chance of conveying benefits, similar conditions must hold in the future. Equivalently, if future conditions are random independently of current conditions, there is no obvious payoff in an adaptation strategy.

3.2 Node Controller Behaviour

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 the node controller is aware, through its interaction with underlying network layers, of all nodes adjacent to the node it resides on, and can communicate with node controllers at neighbouring nodes. As mentioned earlier, in the model, such communication takes place through a buffer at an arc. In addition, each node controller is equipped with a self-loop arc that serves as the default route for messages that cannot immediately be routed to a neighbour node. Since there is no upper bound enforced on communication delays, the node controller always runs the risk that information received from the outside world is out of date.

Node controller behaviour is given in the semantics by the possible reductions on *net* configurations. There are two main kinds of reductions: labelled and unlabelled. Labelled reductions define how the node controller exchanges information with the interpreter layer, while unlabelled reductions define actions that can be taken independently of the interpreter state, such as sending and receiving routing messages. Similarly, interpreter behaviour is given by reductions on *cn* configurations, which are also labelled or unlabelled. The whole state of the system at any time is defined as a pair of a *net* and a *cn* configurations, with a system evolution step being either an independent reduction by one member of the pair or a mutual transition on matching labels. The node controller transition rules related to transmission of routing tables (TABLE messages) are straightforward and unlabelled, with the condition $u' \neq u$ ensuring that node controllers do not send themselves messages through the self loop queue:

$$\textsc{(net\_table\_send)}\quad \frac{u' \neq u \qquad Q \xrightarrow{\text{enqueue}\,(\textsc{Table}\,(\tau))} Q'}{\begin{array}{c}\mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,Q,u')\\ \to \mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,Q',u)\end{array}} \qquad \textsc{(net\_table\_recv)}\quad \frac{Q \xrightarrow{\text{dequeue}\,(\textsc{Table}\,(\tau'))} Q' \qquad \tau \xrightarrow{\text{update}\,(\tau',u')} \tau''}{\begin{array}{c}\mathsf{ar}\,(u',Q,u)\;\mathsf{nd}\,(u,\tau)\\ \to \mathsf{ar}\,(u',Q',u)\;\mathsf{nd}\,(u,\tau'')\end{array}}$$

In contrast, a message *msg* between objects (either a CALL message for a method invocations or a FUTURE messages for a resolved future) requires two labelled rules for sending and receiving it from the object layer, and one unlabelled rule, for routing between nodes when necessary:

$$\textsc{(net\_msg\_recv\_out)}\quad \frac{Q \xrightarrow{\text{dequeue}\,(msg)} Q' \qquad \text{dest}\,(msg) = o \quad o \in \tau}{\begin{array}{c}\mathsf{ar}\,(u',Q,u)\;\mathsf{nd}\,(u,\tau)\\ \xrightarrow{\text{tr}\,(o,msg)} \mathsf{ar}\,(u',Q',u)\;\mathsf{nd}\,(u,\tau)\end{array}} \qquad \textsc{(net\_msg\_send\_in)}\quad \frac{o \in \tau \quad \text{dest}\,(msg) = o' \qquad \text{next}\,(\tau,o',u) = u' \qquad Q \xrightarrow{\text{enqueue}\,(msg)} Q'}{\begin{array}{c}\mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,Q,u')\\ \xrightarrow{\text{tr}\,(o,msg)} \mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,Q',u')\end{array}}$$

$$\textsc{(net\_route\_further)}\quad \frac{Q_1 \xrightarrow{\text{dequeue}\,(msg)} Q_1' \quad \text{dest}\,(msg) = o \quad o \notin \tau \qquad \text{next}\,(\tau,o,u) = u'' \quad Q_2 \xrightarrow{\text{enqueue}\,(msg)} Q_2'}{\begin{array}{c}\mathsf{ar}\,(u',Q_1,u)\;\mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,Q_2,u'')\\ \to \mathsf{ar}\,(u',Q_1',u)\;\mathsf{nd}\,(u,\tau)\;\mathsf{ar}\,(u,Q_2',u'')\end{array}}$$

The condition $o \in \tau$ is to be interpreted as saying that the object o is registered in the routing table τ as being located on the current node. Note also that, when an object sends a message to itself or some other object located on the same node, it will be the case that $u' = u$, and hence the self-loop arc will be used in the reduction. The auxiliary function dest simply returns the first argument in a message, which for the related message types is always the identifier of the destination object.

The rules for moving objects between node controllers are similar to those for passing messages to objects, with the main difference being that no routing is involved for deciding the direction. The rules permit objects being passed around between nodes indefinitely without ever letting the related tasks finish, but implementations will typically want to exclude such executions. Note that both the sender and the receiver needs to update their respective routing tables via auxiliary functions to reflect the new object allocation:

$$\dfrac{\begin{array}{l}(\textsc{net\_object\_send\_in})\\ o \in \tau \quad u' \neq u\\ \tau \xrightarrow{\text{replace}\,(o,u',1)} \tau'\\ Q \xrightarrow{\text{enqueue}\,(\textsc{Object}\,(\textit{object}))} Q'\end{array}}{\begin{array}{l}\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,\,Q,\,u')\\ \overset{\text{mv}\,(\textit{object})}{\rightarrow}\ \mathsf{nd}\,(u,\tau')\,\mathsf{ar}\,(u,\,Q',\,u')\end{array}} \qquad \dfrac{\begin{array}{l}(\textsc{net\_object\_recv\_out})\\ \text{id}\,(\textit{object}) = o\\ Q \xrightarrow{\text{dequeue}\,(\textsc{Object}\,(\textit{object}))} Q'\\ \tau \xrightarrow{\text{replace}\,(o,u,0)} \tau'\end{array}}{\begin{array}{l}\mathsf{ar}\,(u',\,Q,\,u)\,\mathsf{nd}\,(u,\tau)\\ \overset{\overline{\text{mv}\,(\textit{object})}}{\rightarrow}\ \mathsf{ar}\,(u',\,Q',\,u)\,\mathsf{nd}\,(u,\tau')\end{array}}$$

The semantics abstracts from concerns how the routing table τ is concretely represented, specifying its properties only with the help of the auxiliary functions update, next, register and replace. To achieve the possibility of convergence, implementations of these functions must fulfill various postconditions, e.g., that old routes are replaced with new ones in the result of update.

The one remaining reduction rule for node controllers concerns creation of objects, which uses the label rg (for 'registration'):

$$\dfrac{\begin{array}{l}(\textsc{net\_new\_object\_in})\\ \text{fresh}\,(o') \quad o \in \tau\\ \tau \xrightarrow{\text{register}\,(o',u,0)} \tau'\end{array}}{\mathsf{nd}\,(u,\tau) \overset{\text{rg}\,(o,o')}{\rightarrow} \mathsf{nd}\,(u,\tau')}$$

The condition fresh (o') is meant to ensure that the new object identifier is globally unique, which translates to there being no such identifier registered in the routing tables of any node in the network. The condition $o \in \tau$ is needed to ensure that the object o which spawned the new object is actually located on the node in question.

For the sake of an example of how the rules work, assume the object o is located on the node u, and an object with identifier o' is located on the adjacent node u'. Suppose *msg* is a Call message being sent from o to o'. Suppose the queue Q in the arc between the two nodes is initially empty, and that the queue Q' denotes the queue that only contains *msg*. The following reduction sequence then describes how the partial state involving these nodes and the arc between them evolves, when the message is passed from o to o':

$$\begin{array}{rl}\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,\,Q,\,u')\,\mathsf{nd}\,(u',\tau') & \overset{\text{tr}\,(o,\textit{msg})}{\rightarrow}\ \mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,\,Q',\,u')\,\mathsf{nd}\,(u',\tau')\\ & \overset{\overline{\text{tr}\,(o',\textit{msg})}}{\rightarrow}\ \mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,\,Q,\,u')\,\mathsf{nd}\,(u',\tau')\end{array}$$

Another example starting from the same state involves the object *object* with identifier o migrating from u to u'. Let Q'' be the queue that only contains the message Object (*object*). We then get the following sequence of reductions:

$$\begin{aligned}\mathsf{nd}\,(u,\tau)\,\mathsf{ar}\,(u,Q,u')\,\mathsf{nd}\,(u',\tau') &\overset{\text{mv}\,(object)}{\rightarrow} \mathsf{nd}\,(u,\tau'')\,\mathsf{ar}\,(u,Q'',u')\,\mathsf{nd}\,(u',\tau')\\ &\overset{\overline{\text{mv}\,(object)}}{\rightarrow} \mathsf{nd}\,(u,\tau'')\,\mathsf{ar}\,(u,Q,u')\,\mathsf{nd}\,(u',\tau''')\end{aligned}$$

Here, τ'' is the routing table τ updated with the fact that o is now found in the direction of u', while τ''' updates τ' with the fact that o is located on the current node u'.

3.3 Object Behaviour

The reduction rules in the Core ABS semantics which involve only a single object and its internal state have been transferred essentially unchanged into unlabelled interpreter layer reduction rules. For example, the following Core ABS reduction rule:

$$\frac{\text{(COND\_TRUE)} \quad [\![b]\!]_{a\circ l} = \text{True}}{\begin{array}{c}\mathsf{ob}\,(o,a,\{l|\textbf{if}\ b\{\overline{s}\}\ \textbf{else}\ \{\overline{s}'\}\ \overline{sp}\},q)\\ \rightarrow \mathsf{ob}\,(o,a,\{l|\overline{s}\,\overline{sp}\},q)\end{array}}$$

is simply translated into the following rules in ABS-NET:

$$\frac{\text{(ABS\_COND\_TRUE)} \quad [\![b]\!]_{a\circ l} = \text{True}}{\begin{array}{c}\mathsf{ob}\,(o,a,\{l|\textbf{if}\ b\{\overline{s}\}\ \textbf{else}\ \{\overline{s}'\}\ \overline{sp}\},q,Q_{in},Q_{out},\Sigma)\\ \rightarrow \mathsf{ob}\,(o,a,\{l|\overline{s}\,\overline{sp}\},q,Q_{in},Q_{out},\Sigma)\end{array}}$$

In contrast, transitions that involve multiple objects or futures are translated into either transitions involving message passing or labelled rules for exchanging data with the node controller. For object creation, both the node controller rule NET_NEW_OBJECT_IN given above and the following rule for the interpreter layer need to be involved:

$$\frac{\begin{array}{l}\text{(ABS\_NEW\_OBJECT\_OUT)}\\ \text{init}\,(C) = \textit{process}\\ [\![\overline{e}]\!]_{a\circ l} = \overline{v}\\ \text{atts}\,(C,\overline{v},o') = a'\\ \text{forwards}\,(\text{futsof}\,(\overline{v}),o,\{o'\},\Sigma) = \Sigma'\end{array}}{\begin{array}{l}\mathsf{ob}\,(o,a,\{l|x = \textbf{new}\ C(\overline{e});\ \overline{sp}\},q,Q_{in},Q_{out},\Sigma)\\ \overset{\overline{\text{rg}\,(o,o')}}{\rightarrow}\ \mathsf{ob}\,(o,a,\{l|x = o';\ \overline{sp}\},q,Q_{in},Q_{out},\Sigma')\\ \qquad \mathsf{ob}\,(o',a',\textbf{idle},\textit{process},(),(),[\,])\end{array}}$$

When such a mutual transition has taken place, the new object has been properly added to the interpreter layer, and its globally unique identifier registered on the node of the object that spawned it. The init and atts auxiliary functions are unchanged from Core ABS, and construct the initial task of the object as given in the corresponding class definition, and initializes variables based on given arguments, respectively. The auxiliary function forwards produces an update to the forwarding obligations in Σ to include futures in the argument list.

Given that the semantics abstracts from details on marshalling and just passes object states directly in messages, the interpreter layer rules for object mobility, which interact with the rules NET_OBJECT_SEND_IN and NET_OBJECT_RECV_OUT above, are very straightforward:

$$\frac{(\textsc{abs\_object\_send\_out})}{\{object\ cn\} \overset{\overline{\text{mv}\,(object)}}{\rightarrow} \{cn\}} \qquad \frac{(\textsc{abs\_object\_recv\_in})}{\{cn\} \overset{\text{mv}\,(object)}{\rightarrow} \{object\ cn\}}$$

Although the rules for actually generating object-addressed messages are relatively complex due to the forwarding of futures, the rules for passing messages back and forth with the node controller are uncomplicated since eligible messages have been put in the out queue of the object:

$$\frac{\begin{array}{c}(\textsc{abs\_msg\_send\_out})\\ Q_{out} \xrightarrow{\text{dequeue}\,(msg)} Q'_{out}\end{array}}{\begin{array}{c}\mathsf{ob}\,(o,\,a,\,p,\,q,\,Q_{in},\,Q_{out},\,\Sigma)\\ \overset{\overline{\text{tr}\,(o,msg)}}{\rightarrow}\ \mathsf{ob}\,(o,\,a,\,p,\,q,\,Q_{in},\,Q'_{out},\,\Sigma)\end{array}} \qquad \frac{\begin{array}{c}(\textsc{abs\_msg\_recv\_in})\\ Q_{in} \xrightarrow{\text{enqueue}\,(msg)} Q'_{in}\end{array}}{\begin{array}{c}\mathsf{ob}\,(o,\,a,\,p,\,q,\,Q_{in},\,Q_{out},\,\Sigma)\\ \overset{\text{tr}\,(o,msg)}{\rightarrow}\ \mathsf{ob}\,(o,\,a,\,p,\,q,\,Q'_{in},\,Q_{out},\,\Sigma)\end{array}}$$

4 Adaptation

We consider three QoS objectives which runtime adaptation solutions can be assessed against: 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, rather than something extrinsic to our model such as processor load or the `loadavg` measure available in many Unix operating system variants. We need a model-intrinsic measure of load to enable reasoning at an abstract level about convergence to balanced allocations 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. 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.

4.1 Node Load Balancing

Although we wish to simultaneously meet all 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. [3] 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 [11] 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 knowing precisely all loads in the system, which is complicated and costly to find out in the current setting.

Berenbrink et al. [4] 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, e.g., via synchronizers [2], to a fully asynchronous setting. 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. If the network graph is sparse, and the number of active tasks an order magnitude greater than the number of nodes, allocations where the difference in load between any two neighbours is one but the maximal load difference is in the order of the graph diameter are possible. Such allocations clearly cannot be improved upon without neutral moves.

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 made worse by information becoming stale, which is a fact of life in asynchronous systems. If the information is not *too* stale, however, the number of oscillation periods can sometimes be bounded [12].

4.2 Minimizing Communication and Other Objectives

The literature on load balancing related to scientific computing contains work on simultaneously optimizing task allocations and communication overhead. For example, Cosenza et al. [7] 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, objects can communicate whenever object identifiers are known to the sender, making it harder to minimize communication overhead. Catalyurek et al. [5] 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, immediate knowledge of the data and computations on each node.

Querying the load of neighbours before deciding where to migrate an object can be costly in terms of arc load, and information received previously may not be accurate. Many load balancing algorithms therefore have as a feature that the number of load queries sent is minimal when migrating a resource. A third measure which is discussed in the literature which we do not consider is the cost in terms of time and messaging for migration itself.

5 Evaluation

In addition to the theoretical results on ABS-NET described elsewhere [8, 9], we have evaluated ABS-NET by developing a simulator for running ABS programs in a network of nodes according to our semantics. We have run the simulator with a variety of network node topologies, programs and object migration policies.

5.1 Simulator

Our simulator's main purposes are to serve as a proof-of-concept for ABS-NET, and to allow us to run various adaptability case studies with particular programs and topologies. Specifically, we are interested in studying convergence properties of object migration policies in practice, and in showing that our approach of distributed execution scales to networks with many nodes. There are several other ways of executing ABS programs developed in the HATS project [10], but the main feature we need that is absent from all of them is object mobility between nodes or sites. Also, in contrast with most of these ABS backends, which aim to provide an execution platform for the full ABS language, the simulator only supports a subset of the Core ABS language; notably, the `await` statement is not supported.

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 [16]. One reason for choosing to use 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 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 broadcast to neighbours after entries have been changed or added as a result of statement execution or incorporation of routes from neighbour messages. Hence, except after a short interval with many updated locations, we expect routing tables to be up-to-date or nearly so. The node controller also stores incoming messages that cannot be processed locally or rerouted.

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 policy through the CLI, which is assumed to be the same for all node controllers in the network. A migration policy is based on one of the adaptation strategies described below.

By default, the simulator starts the initial task of the initial object on a single startup node. In all our programs, the initial 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 proper distributed garbage collection, which we have not implemented.

One desirable feature that the simulator currently lacks is control of link characteristics, such as delays.

5.2 Scenarios

There are many parameters to consider when setting up interesting scenarios for studying adaptation via simulations, as outlined below.

Network configuration The size and topology of the network. Large and dense networks obviously give more overhead in the form of messaging (e.g. routing and load), making simulations slower.

Object behaviour The number of objects generated by the program, inter-object communication patterns, and the fraction of objects with active tasks over time. In practice, this means selecting the appropriate ABS program and adjusting some method parameters.

Adaptation strategy This includes both the logic for deciding when and where to migrate objects, and for messaging to exchange information used as basis for decisions.

By necessity, we have explored only a small cross-section of the possible parameters, at this initial stage of the work.

5.2.1 Network Configurations

The possible sizes and configurations of networks to be simulated are limited by the performance of the prototype simulator. 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 for nodes along the continuum 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 for some scenarios to compare results.

5.2.2 Benchmark Programs

We have developed a number of ABS programs specifically to run in our simulator. All programs share the characteristic that they 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 a distributed hash table (DHT) algorithm, communication patterns among generated objects follows 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. After running initial simulations, we have adjusted parameters in our programs, and in some cases added functionally redundant instructions, to get constant and reasonably 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 over time becomes hard to discern. Sometimes this is due to behaviour inherent to the program, as in the convergecast program described in Section 2, which gives rise to periodic bursts of messages. We focus on programs with more consistent behaviour.

`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 simply a completely even distribution of objects to nodes, regardless of the network topology.

`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.

`Star.abs` An object star configuration consists of a 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 of object star configurations.

`ChordDHT.abs` An implementation of the Chord DHT algorithm [20]. Key-value mappings are distributed between a number of nodes, which all support a put/get interface to clients. Nodes are arranged in a ring, but aside from references to their neighbours, each node has $\log(n)$ "fingers" to non-adjacent nodes, where n is the size of the keyspace. The addition, or join, of a node to the ring places the new node at a particular position based on its identifier and can trigger global reconfiguration of the ring. During setup, 128 nodes are joined to the chord, and each node 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.

5.2.3 Adaptation Strategies

We have generally restricted ourselves to strategies that as a first priority balance out load evenly among nodes in the network. As a consequence, a simulated node controller continually informs neighbours 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 policy defines a callback method which takes the affected node controller as a parameter. The callback method is invoked, and can possibly result in the migration of several objects to neighbour nodes.

Berenbrink et al. An adapted version of the selfish distributed load balancing algorithm by Berenbrink et al., 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 restrictions in the size of message buffers.

Berenbrink et al. with neutral moves An adapted version of the selfish distributed load balancing algorithm by Berenbrink et al., which does allow neutral moves, and therefore is only expected to converge to a completely stable state after a long time, exponential in the size of the network. As determined experimentally, only migrating one or two objects per cycle leads to significantly less oscillation of objects than when directly implementing the abstract description given in Algorithm 2.

Berenbrink et al. with communication intensity A variant of the preceding policy, where objects are 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. Specifically, if the load difference is d, the probability of migration becomes $\frac{d}{10}$, adjusted to closest number in the interval $[0, 1]$.

Weighted neighbour load difference with communication Given a randomly chosen object and adjacent node as in the previous policy, 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 policy. 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
  $u'$ is a neighbour chosen uniformly at random
  $l$ is the current load
  $l'$ is the last known load of $u'$
  if $l > l' + 1$ then
    send $o$ to $u'$ with probability $1 - l'/l$
  end if
end for
```

Algorithm 2 Berenbrink et al. load balancing with neutral moves cycle.

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

5.3 Scenario Objectives

Since our primary objective is to balance node load evenly, we record the load of all individual nodes over time, and then show 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. The imposed fairness provides a degree of synchrony in the simulated network.

5.4 Results

Below, we give an overview of the results from our simulations of the scenarios described in the previous sections.

5.4.1 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. 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 case of a 32-node grid, the stable allocation was in almost all cases some distance 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 but five migration 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.

The maximum load and the standard deviation of the load over time for the 32-node hypergraph network topology is shown in Figure 3 and Figure 4, respectively. The corresponding measures over time for an 8×4 grid topology are shown in Figure 5 and Figure 6, respectively. For a grid, the gain from using neutral moves is most distinctly recognized in the lower standard deviation compared to the algorithm without neutral moves in Figure 6. Results for running the program on a 32-node complete network graph are essentially the same as for the hypergraph case, and therefore omitted.

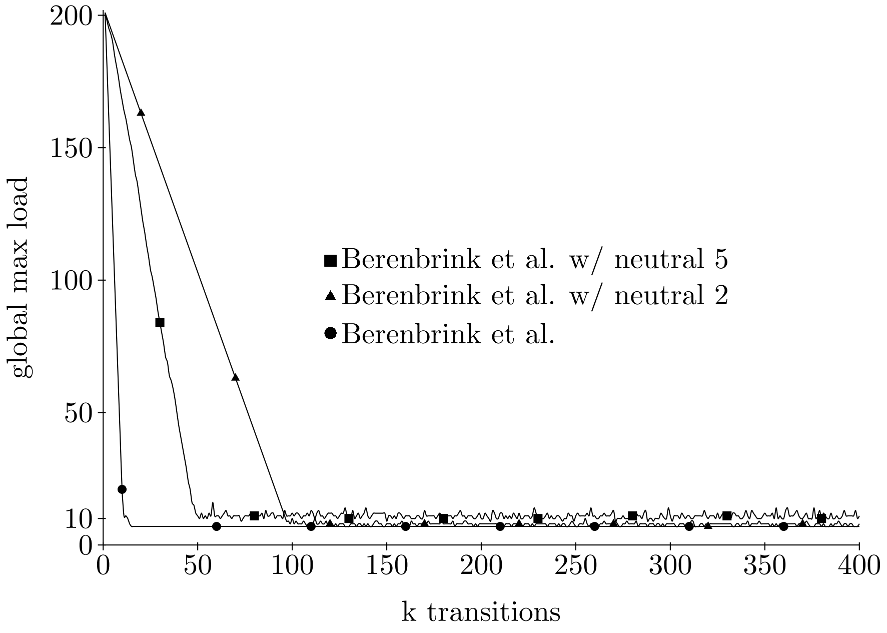

Figure 3: Global max load in hypergraph for `IndependentTasks.abs`.

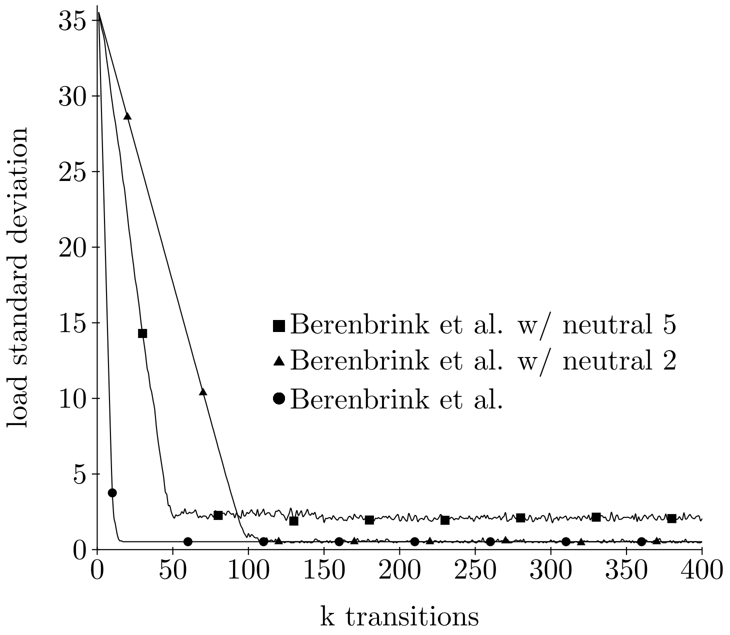

Figure 4: Load std. deviation in hypergraph for `IndependentTasks.abs`.

5.4.2 Simulations of `Star.abs`

In the star program, we construct stars precisely so that each node can hold a whole star, and there is precisely one node per network node. In an

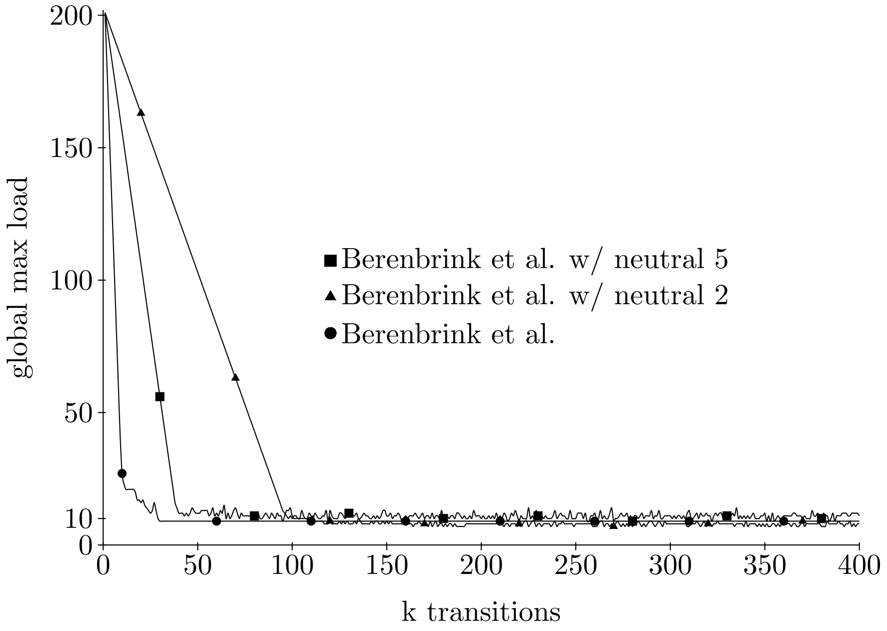

Figure 5: Global max load in grid for `IndependentTasks.abs`.

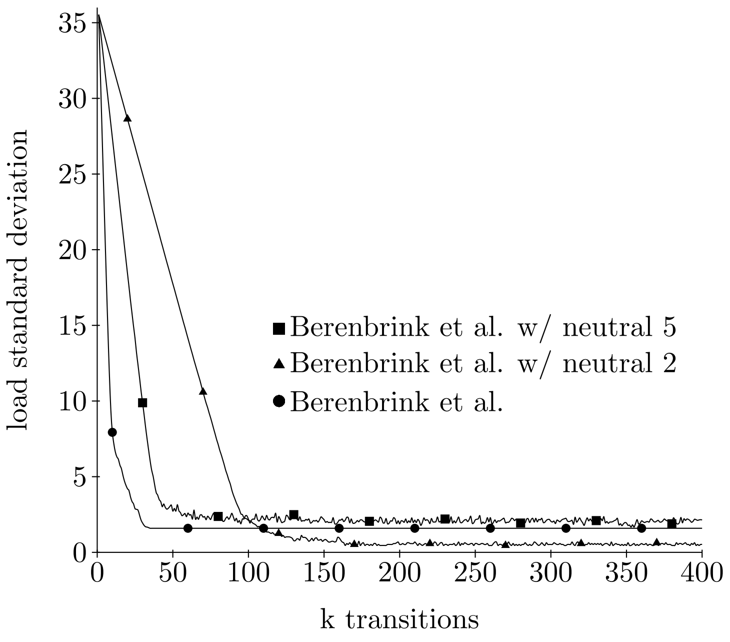

Figure 6: Load standard deviation in grid for `IndependentTasks.abs`.

optimal allocation, therefore, there are no node-to-node message exchanges at all; all messages are sent locally.

We expected the pure load balancing policies to have markedly worse results than the policies taking inter-object communication intensity into account. In Figure 7, the standard deviation of the number of sent messages over time is shown. In Figure 8, the average number of sent messages over time is shown. In both cases, the measurements have been smoothed out via averaging over five samples to reduce noise. As can be seen in the figures, 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 policies, 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 the hypergraph, load became evenly balanced relatively quickly, as seen in Figure 9, similarly to hypergraphs when running `IndependentTasks.abs`. Hence, there was no significant avoidance of messaging by communicating objects clustering at a few specific nodes.

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 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 Figure 10, gray bars show the distribution of total distance among the 32 center objects 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 distinctly worse, with most centers in the 10–20 distance range, while the latter algorithm has most centers in the 6–12 distance range.

5.4.3 Simulations of `Ring.abs`

When running a ring of 129 objects on a 32-node grid, there are balanced allocations where all but one node have 4 objects, where all objects that communicate are on either 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. Such allocations lead to few inter-node messages being needed for a method invocation that involves the whole ring. Almost all objects then have a combined distance of one to the objects they communicate with.

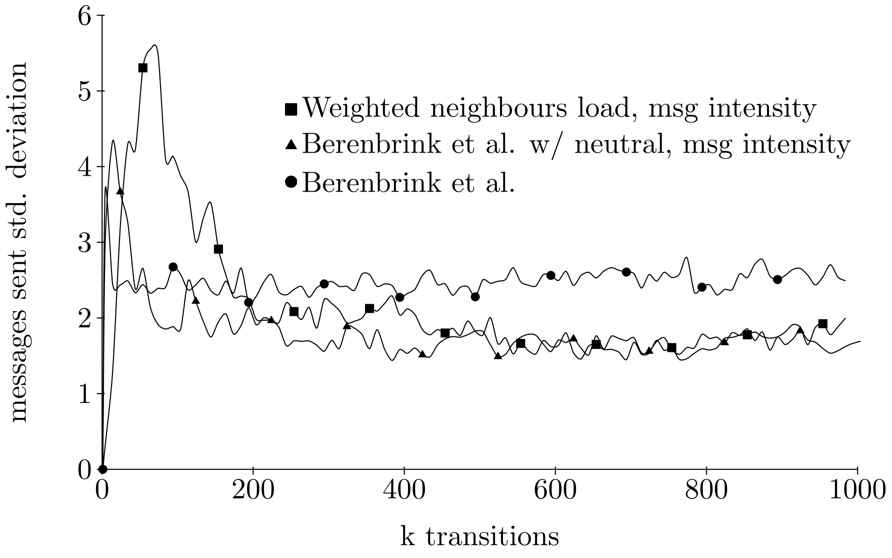

Figure 7: Std. deviation of sent messages for hypergraph in `Star.abs`.

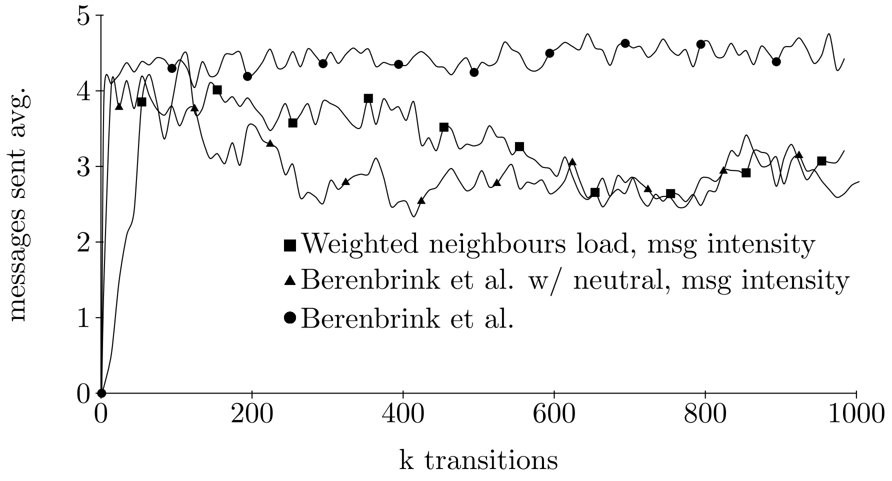

Figure 8: Avg. sent messages for hypergraph in `Star.abs`.

In Figure 12, the standard deviation of messages of a 129-object ring for a grid topology is shown. Here, both the solutions which take message intensity into account show considerable improvement over time. This is also reflected in the average number of messages sent over time shown in Figure 13. The eventually lower number of messages sent are not due to clustering of many objects on a few nodes, as shown by the eventually low standard deviation for all migration strategies in Figure 11.

In Figure 14, gray bars show the distribution of total distance among all ring objects on a grid to the objects they communicate with, after 1000

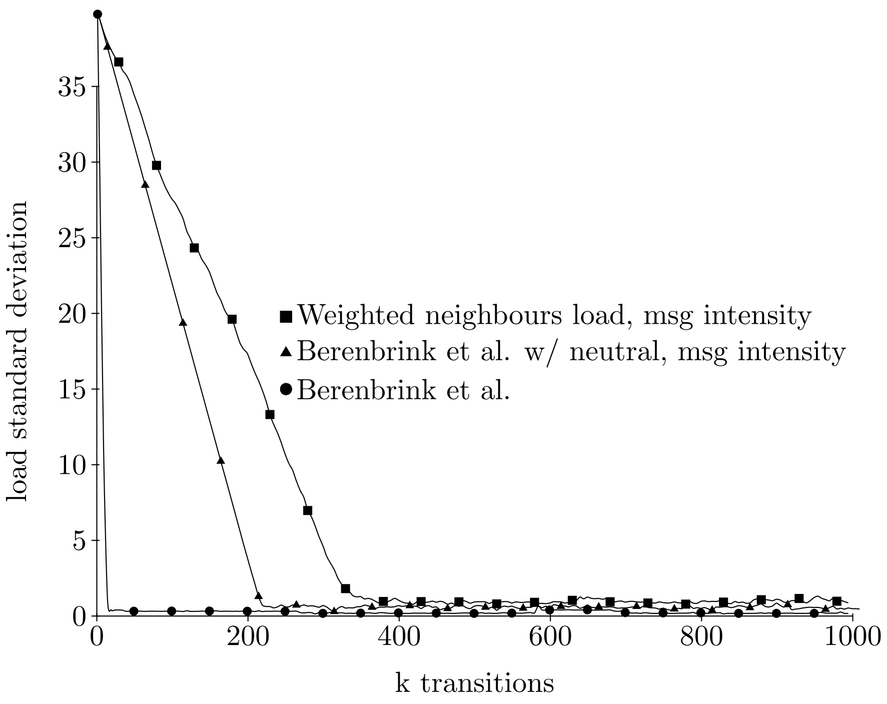

Figure 9: Std. deviation of load for hypergraph in `Star.abs`.

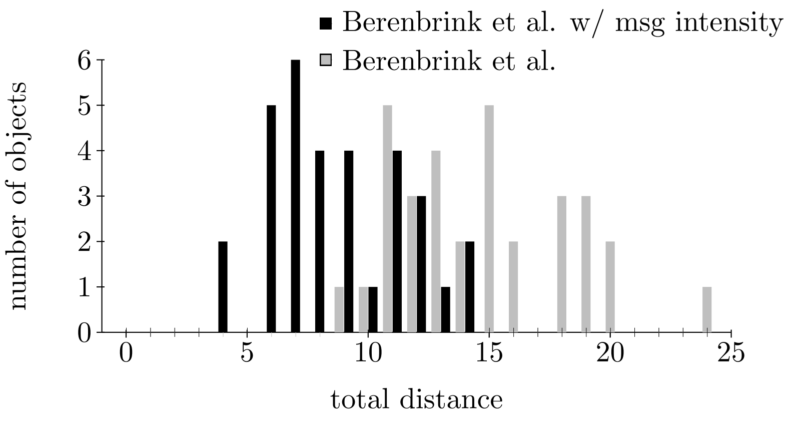

Figure 10: Object distance distribution for hypergraph in `Star.abs`.

migration cycles using the algorithm by Berenbrink et al. Black bars show the distribution for the algorithm by Berenbrink et al. 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.

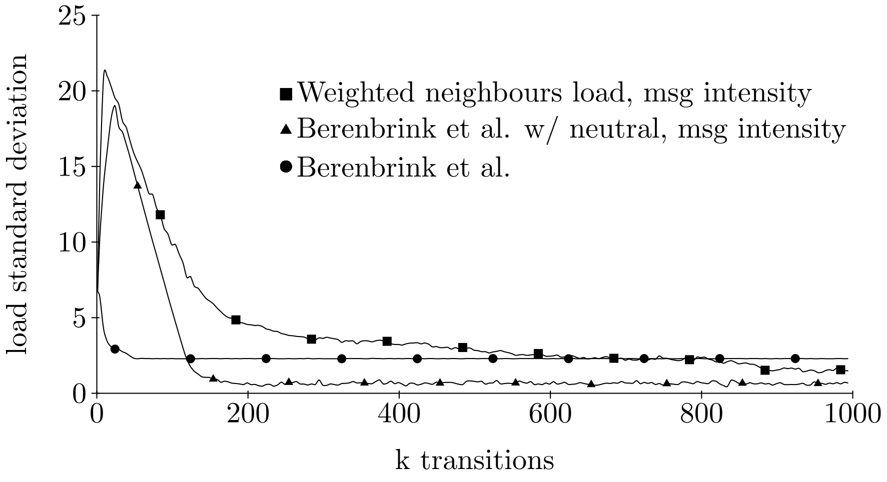

Figure 11: Std. deviation of load for grid in `Ring.abs`.

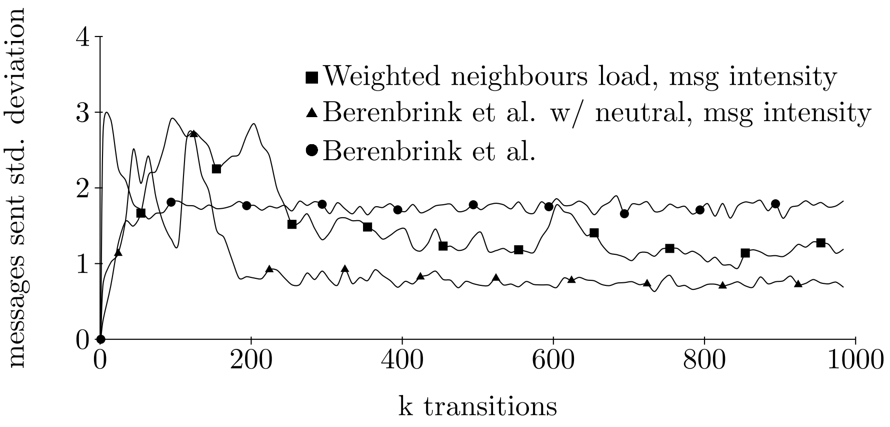

Figure 12: Std. deviation of sent messages for grid in `Ring.abs`.

5.4.4 Simulations of `ChordDHT.abs`

In the setup phase, a number of `ServiceObject` nodes are created and joined. Then, every such object becomes associated with either a producer object, which puts values into the DHT via the `put` method, or a consumer object, which tries to retrieve them via the `lookup` method.

Figure 15 shows the standard deviation of the number of messages sent for nodes when running the program on a hypergraph, and Figure 16 shows the average number of messages. In both figures, the curves have been smoothed out by averaging samples five at a time. The weighted neighbours load and message intensity strategy exhibited a tendency to quickly cause message buffer overflows for this program, which is why we do not show any results for it. The figures confirm that there is a reasonable payoff from

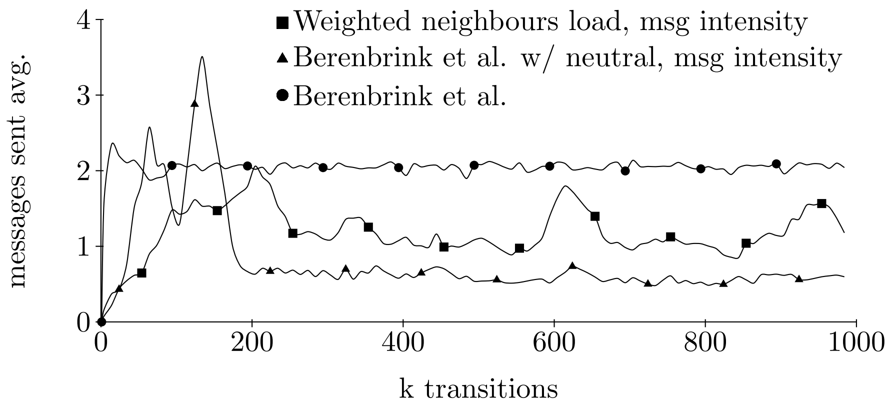

Figure 13: Avg. sent messages for grid in `Ring.abs`.

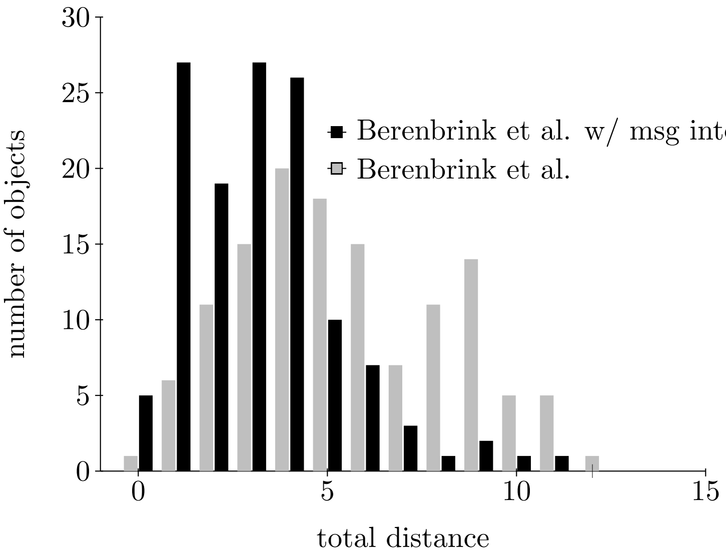

Figure 14: Object distance distribution for grid in `Ring.abs`.

taking messaging into account in a migration strategy, even when running a program with relatively complex communication patterns.

5.4.5 Simulations of `Star.abs` on 64-Node Grid

We expected the improved performance of algorithms which consider message intensity to be greater on larger networks. We therefore investigated how the star program performed on a 64-node grid, with the number of star centers scaled up appropriately. The resulting program is called `Star64.abs`. The results were largely as expected, with a larger discrepancy between the pure load balancing procedure as compared to procedures which take mes-

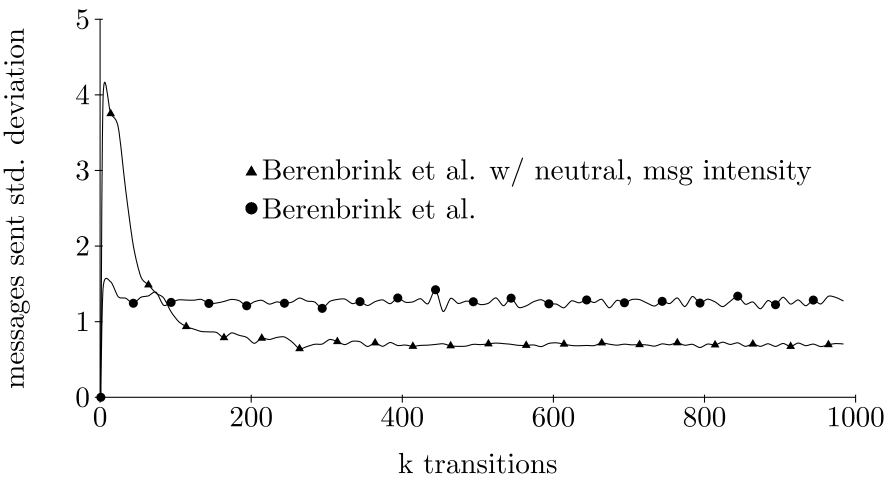

Figure 15: Std. dev. of sent messages for hypergraph in `ChordDHT.abs`.

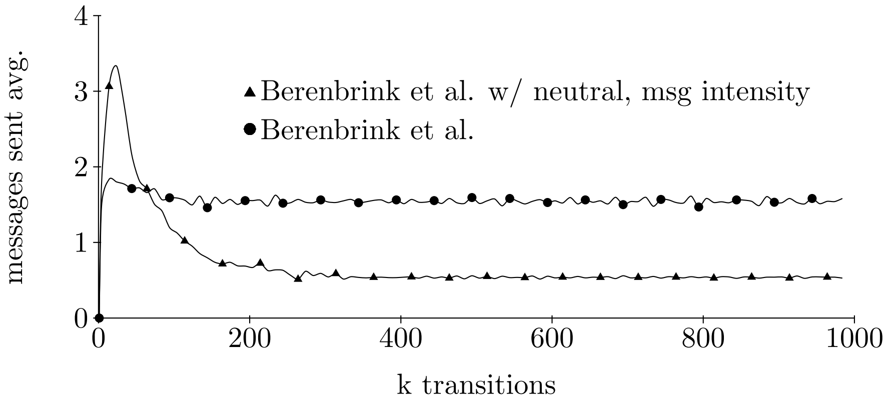

Figure 16: Avg. sent messages for hypergraph in `ChordDHT.abs`.

saging into account, as shown in Figure 17 and Figure 18.

6 Conclusions and Future Work

The evaluation suggests 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 main concern for our results being valid for real-world networks is the use in our network 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 man-

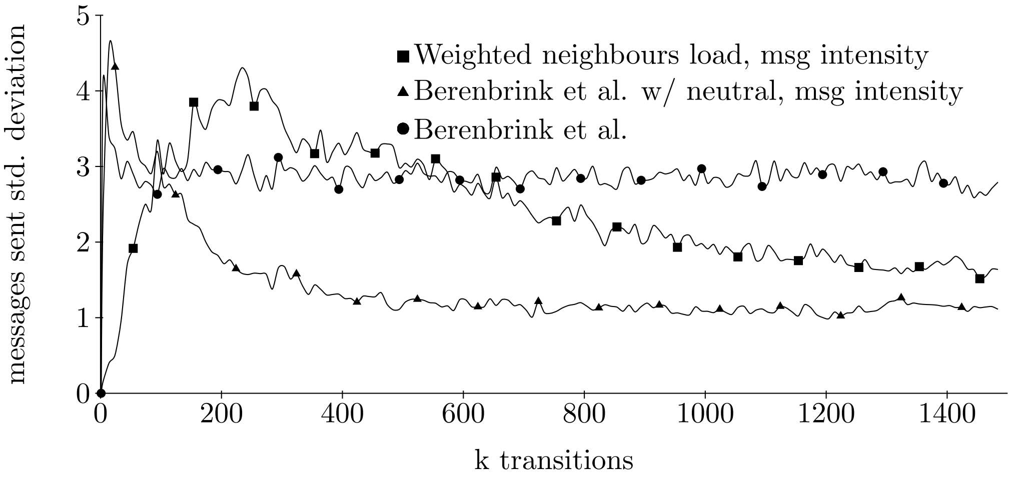

Figure 17: Std. dev. of sent messages for grid64 in `Star64.abs`.

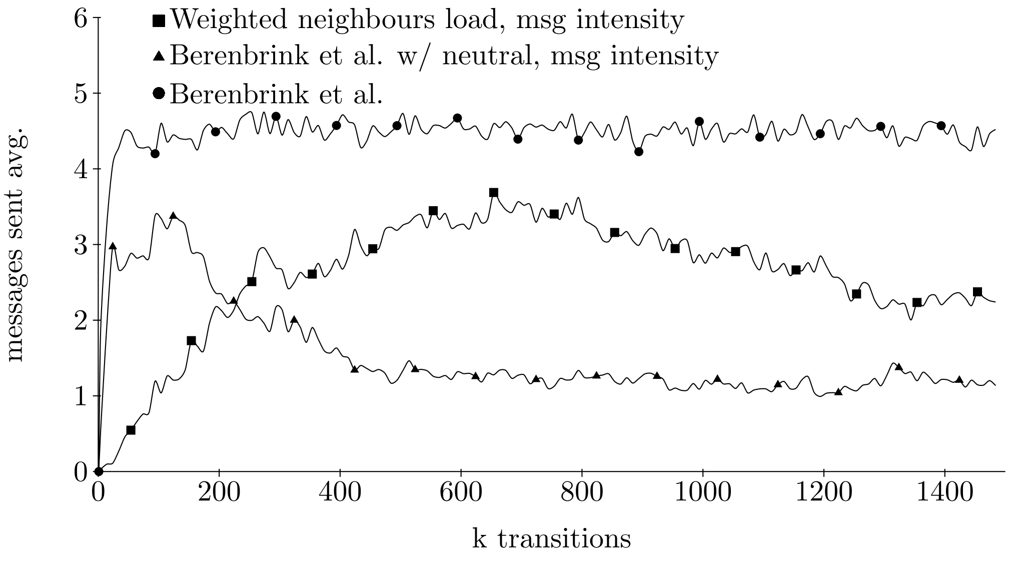

Figure 18: Avg. sent messages for 64-node grid in `Star64.abs`.

agement, 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, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In *Proceedings of the 17th international conference on Formal methods*, FM'11, pages 353–368, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, Oct. 1985.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *SIAM Journal on Computing*, pages 593–602, 1994.

[4] P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. W. Goldberg, Z. Hu, and R. Martin. Distributed selfish load balancing. In *In Proc. 17th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA*, pages 354–363. ACM Press, 2006.

[5] U. V. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proceedings of 21st IEEE International Parallel & Distributed Processing Symposium*, 2007.

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

[7] B. Cosenza, G. Cordasco, R. De Chiara, and V. Scarano. Distributed load balancing for parallel agent-based simulations. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 62 –69, feb. 2011.

[8] M. Dam. Location independent routing in process network overlays. Manuscript, to be submitted.

[9] M. Dam and K. Palmskog. Efficient and fully abstract routing of futures in object network overlays. Manuscript, to be submitted.

[10] ABS Tool Platform and Tutorial, Mar. 2013. Deliverable 1.5 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`. To appear.

[11] E. Even-Dar and Y. Mansour. Fast convergence of selfish rerouting. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 772–781, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

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

[13] FP7-231620 (HATS) Project. Deliverable 1.2: Full ABS modeling framework, March 2011. `http://www.hats-project-eu`.

[14] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of Software Technologies Concertation on Formal Methods for Components and Objects*, 2010.

[15] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

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

[17] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

[18] P. Sewell, P. T. Wojciechowski, and A. Unyapoth. Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Transactions on Programming Languages and Systems*, 34, April 2010.

[19] M. Sirjani and M. M. Jaghoori. Ten years of analyzing actors: Rebeca experience. In G. Agha, J. Meseguer, and O. Danvy, editors, *Formal modeling*, pages 20–56. Springer-Verlag, Berlin, Heidelberg, 2011.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.