

Copyright

by

Kristi Morton

2008

Orc-X: Combining Orchestrations and XQuery

by

Kristi Morton, B.A.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

May 2008

Orc-X: Combining Orchestrations and XQuery

Approved by
Supervising Committee:

Acknowledgments

This thesis wouldn't be possible without the help of my committee advisors, William Cook and Jayadev Misra, or my remote advisors at AT&T Labs, Mary Fernandez and Trevor Jim. I would like to thank Mary and Trevor for helping me form the idea of extending Orc with XQuery and I thank William and Jayadev for providing helpful insight into the Orc language and implementation. Additionally, I thank Dave Kitchin for his help during the formative stages of the implementation of Orc-X.

Most importantly, I thank my husband, Robert Morton, for all of his support and patience with me during stressful times.

KRISTI MORTON

The University of Texas at Austin

May 2008

Orc-X: Combining Orchestrations and XQuery

Kristi Morton, M.A.

The University of Texas at Austin, 2008

Supervisor: William Cook

In designing a coordination language for distributed computing, it is desirable to have explicit concurrency coupled with a data model that supports the communication requirements of the distribution model. Orc is a language that provides simple but powerful concurrency constructs to orchestrate distributed computations in the face of failures. Since Orc's communication model is based on web services, it is natural to consider XML as an appropriate data model for Orc. We present Orc-X, an extension of the Orc language with an XML data model and XML-specific data management capabilities from XQuery. We demonstrate that Orc-X is well-suited for highly concurrent, distributed resource management protocols such as Narada that manage ad hoc, peer-to-peer networks.

Contents

Acknowledgments	iv
Abstract	v
List of Tables	viii
List of Figures	ix
Chapter 1 Introduction	1
Chapter 2 Background	6
2.1 Orc overview	6
2.2 XQuery	11
2.3 Narada	12
Chapter 3 Orc-X language overview	14
3.1 Syntax and semantics	14
3.2 Orc-X implementation	16
Chapter 4 Experimental evaluation	25
4.1 Narada implementation in Orc-X	25
4.2 Evaluation	35

Chapter 5 Related work	49
Chapter 6 Conclusions and future work	54
Appendix A Narada in Orc-X	57
Bibliography	68
Vita	71

List of Tables

4.1	HPROF CPU samples from 4 nodes running Narada	38
4.2	HPROF CPU time (in ms) from 4 nodes running Narada	39
4.3	Cache hit rates for 4 nodes running Narada	40
4.4	Execution time (in ms) for 4 nodes running Narada	40

List of Figures

3.1	Orc-X extension of Orc syntax	15
3.2	Orc-X architecture: All inter-peer/server communication is through HTTP	17
3.3	Flash Player GUI with four Orc engines running Narada	23
4.1	Before (top) and after (bottom) initialization	44
4.2	Members overlay tree: initial (top) and final (bottom)	45
4.3	Routes overlay tree: initial (top) and final (bottom)	46
4.4	Members and routes trees after node 3105 dies	47
4.5	Route overlay tree with long latency between 3102 and 3101	48

Chapter 1

Introduction

This thesis presents Orc-X, an extension of the Orc language with an XML data model and XML-specific data management capabilities from the XMLQuery (XQuery) language. Orc [19] is a language that provides simple but powerful constructs to orchestrate distributed computations in the face of failures. However, Orc lacks constructs for managing data, which are necessary for building complete applications. Orcs communication model is based on web services, which support complex XML documents in addition to simple data types. Thus it is natural to consider XML as an appropriate data model for Orc. XQuery [3] is declarative query language for XML-based applications. We demonstrate that Orc-X is well-suited for the application domain of distributed resource management protocols such as the Narada [7] mesh overlay protocol. We hypothesize that XML is a natural data model for the Orc language and, together they will help realize structured concurrent programming.

The Orc-X project has an immediate goal of simulating a complex distributed resource protocol, which will serve as our motivating application and basis of evaluation. Narada is a distributed protocol that manages the structure of an overlay-network. Furthermore, Orc-X aims to bring an appropriate data model to the Orc

language. The original design of the Orc language focused on concurrency constructs and included only basic scalar data types. The data model was considered orthogonal to the main intent of Orc, which is concurrency and communication. However, without a data model it is difficult to write large applications or interface with complex services. We have chosen to extend Orc by embedding support for the XML [5] data model. XML has proven to be a robust data model for communication in many wide-area systems by supporting complex structured data models that can be extended as the system evolves. Since Orc is a communication-centric language, the internal data model should fit well with the communication model. Furthermore, Orc's intended use is in orchestrating web services, which rely on XML for messaging and representing data content. Finally, XML is a natural fit for Orc because it is easy to serialize and ship XML data across a network.

Orc-X marries two programming paradigms together: a concurrency and communication-centric language (Orc) with a declarative query language (XQuery). The Orc language is a domain-specific language for distributed and concurrent computation [19]. Orc provides three combinators that organize and structure the flow of a concurrent computation, similar to the way *structured programming* [8] organizes sequential program flow. *Structured concurrency* [14] disallows arbitrary message passing and constrains communication and synchronization, in the same way that *structured programming* [8] disallows arbitrary jumps and constrains program control flow. Because of these restrictions, Orc's combinators obey strong algebraic laws, and Orc programs are easier to reason about locally. XQuery, on the other hand, is a query-based language targets XML-based applications. XQuery has specific language features, for e.g. XPath [11] expressions, that navigate and query XML documents. XQuery also has update semantics which allow for updating XML documents [6].

Current language research for applications in distributed resource manage-

ment protocols has focused on potential of declarative query languages such as XQuery through the Distributed XQuery (DXQ) work [10, 9] on expressing the Domain Name Server (DNS) and Narada protocols. There also has been recent work on using declarative logic languages such as OverLog [15] and NDlog [16] to express declarative networking protocols such as Narada. Both declarative query and logic languages have shown promise in efficiently expressing the semantic needs of these applications over general-purpose languages. On the other hand, the implementation of distributed resource management protocols in general-purpose languages often uses a rigid, ad-hoc data model and suffers from code bloat. For example, the comparison of two implementations of the DNS protocol (in DXQ and C) given in [9] indicates that the relative code size is an order of magnitude larger in the C language: in DXQ its less than 100 lines of code, while in the C language its implemented in 5000+ lines of code.

In summary, the expressive power of declarative languages like DXQ and OverLog comes from leveraging a relational or semi-structured data model which fits well with the declarative programming style. The Orc-X language combines the best of both worlds: structured concurrency through the Orc language with an XML data model that is efficiently managed by XQuery, a declarative query language. Structured concurrency gives programmers a decreased reliance on synchronization primitives and makes it easier to write and debug code. Finally, Orc provides explicit primitives for time-outs or failure of services. Because the notion of time is built into the language [22], Orc-X allows a programmer to easily express time-outs in the case of a failure: for example in a dropped communication link between nodes.

Modern applications increasingly require support for concurrent computation and management of complex, distributed data. For example, the emerging area of grid computing has led to increasing focus on applications that manage distributed resources to create overlay networks or other network services. Job scheduling is a

typical grid computing problem, which requires query capabilities, distribution and the ability to communicate with other nodes in the network. One distributed protocol that we chose to focus on is the Narada protocol. This protocol sends periodic, asynchronous messages around an overlay network to manage its structure [7]. The structure of the overlay network is encoded in the node-local state, which includes neighbor and group memberships as well as routing tables [7]. This information is naturally represented in XML and we use XQuery to query and manage the semi-structured XML data at each node. Because the protocol sends concurrent messages that probe for changes to the overlay network, Orc-X is well-suited to orchestrate the concurrent invocation of these messages, while managing time-outs and communication failures that inevitably arise in distributed environments. Finally, Narada demonstrates the potential of Orc's structured concurrency operators to concisely and elegantly express distributed computations. These applications must be resilient to failure of nodes and communication. Such applications operate in a dynamic environment, in which nodes join and leave the network. Current programming languages have some inherent weaknesses in building these kinds of distributed applications. In summary, general purpose languages are inefficient, declarative languages like DXQ have synchronization issues, logic languages like OverLog are hard to read and understand and Linq lacks explicit concurrency primitives.

The intent of this thesis is to satisfy two immediate goals of Orc-X: to bring the idea of structured concurrent programming to fruition and to demonstrate that XML is an appropriate data model for Orc. We use the Narada protocol as a motivating application to evaluate how well we achieve these goals.

The rest of this thesis is devoted to describing the Orc-X language and our working system. In Chapter 2, we discuss how Orc and XQuery work in greater detail. Furthermore, we discuss the main features of the Orc-X architecture, showing how Orc-X works as a language and providing rationale regarding the design deci-

sions of the language and our infrastructure. Chapter 3 reports on our experience implementing the Narada protocol in Orc-X. In Chapter 4 we discuss the feasibility and performance of Orc-X as a language in representing distributed resource management protocols. Chapter 5 reports how Orc-X differs from other recent work on domain-specific programming languages. We conclude our work on Orc-X in Chapter 6 and discuss future work and improvements.

Chapter 2

Background

The Orc language is presented by example, starting with a simple weather forecast example and progressing through more complex examples involving programming idioms used in the Narada implementation. Furthermore, we present a brief introduction to the XQuery language and a high-level overview of the Narada protocol.

2.1 Orc overview

Orc is an executable process calculus that expresses structured concurrency. There are three main operators for composing expressions: parallelizing, sequencing and selective pruning. In Orc, *sites* are responsible for performing data operations such as computation and communication. A site call can represent a function call or web service invocation, for example, and may return at most one result. If it never returns a result, we say it is *silent*. In the following examples we use the site call `Weather(ac)` as a web service that provides weather forecasts for the given airport code `ac`, and the site call `Email(a,m)` as a function that sends a message `m` to address `a`.

The sequential composition defines an ordering between two expressions,

written as: `f >x> g` or simply `f >> g`. The `x` in the first sequential composition is bound to the value published by `f`, while the second sequential composition has no variable binding. The following example invokes the `Weather` web service, binds the result to a message `m` and e-mails the message to address `a`.

```
Weather("AUS") >m> Email(a,m)
```

The parallel composition allows two expressions `f` and `g` to be composed as: `f | g`. Here `f` and `g` are treated as independent computations and executed asynchronously in parallel. Upon termination of both expressions there may be up to two results published. The results are published as soon as a computation terminates, and no result is published if the computation never terminates. In the following example there could be zero, one or two weather reports published depending on the responsiveness of the `Weather` services for Austin and Barcelona. Upon each publication `Orc` will create a new thread for invoking `Email` with `m` bound to the published web report within the scope of the new thread.

```
(Weather("AUS") | Weather("BCN")) >m> Email(a,m)
```

The asymmetric parallel composition is used for selective pruning, which is written as `f where x in g`. This construct starts executing `f` and `g` in parallel. Any subexpression within `f` that is dependent upon `x` will suspend executing until `x` has a value bound. When `g` has published a value, `x` is bound to the result and any subexpressions within `g` that are still executing are terminated. In the following example we approximate the weather in Texas by selecting the first response from any of the major cities.

```
Email(a,m) where m in  
(Weather("IAH") | Weather("DFW") | Weather("SAT"))
```

`Orc` defines several fundamental sites for key programming constructs. The site `let(x,y,...`) will explicitly publish a tuple of values. In contrast, the site

`Signal` causes the publication of a signal with no data value. As with any publication this can be used to trigger a sequence of operations. The site `Rtimer(t)` publishes a signal after `t` time units. The site `if(b)` publishes a signal when `b` is true, but remains silent otherwise. The following example demonstrates the use of signaling in a sequence.

```
if(Weather.isHumid("IAH")) >> Email(a, "Dress lightly")
```

Additionally Orc allows expressions to be defined with a name and optionally with parameters using `def`. An expression can be invoked much like a site, however it may publish any number of values. The following example defines a `Metronome` expression that will publish a signal every unit of time, indefinitely. The `Metronome` invocation is sequenced into a request for a weather report, causing the recipient to receive regularly timed updates.

```
def Metronome =  
  Rtimer(1) >> Signal | Metronome  
  
def WeatherUpdates(ac) =  
  Metronome >> Weather(ac) >m> Email(a,m)
```

The Orc operators are arranged in increasing order of precedence as: `def`, `where`, `|`, `>x>`. The operator `>x>` is right-associative, the operator `where` is left-associative, and the operator `|` is fully associative and commutative.

Fork-join Parallelism

In concurrent programming, one often needs to spawn two independent threads at a point in the computation, and resume the computation after both threads complete. Such an execution style is called *fork-join* parallelism. There is no special construct for fork-join in Orc, but it is easy to code such computations. The following code

fragment calls sites M and N in parallel and publishes their values as a tuple after they both complete their executions.

```
(let(u,v) where u in M) where v in N
```

Time-out

Distributed resource management protocols interact in environments that exhibit dynamic behavior. For example, a ping request to a remote peer may not return a result if the peer’s communication link is down. We show in section 4.1 how time-outs are used to mitigate this problem. Because Orc has time built into its semantics [22], it is easy to express time-outs. The expression

```
let(z) where z in (f | Rtimer(t) >> let(3))
```

either publishes the first publication of f , or times out after t units and publishes 3. A typical programming paradigm is to call site M and publish a pair (x,b) as the value, where b is true if M publishes x before the time-out, and false if there is a time-out. In the latter case, x is irrelevant. Below, z is the pair (x,b) .

```
let(z) where z in
  ( M >x> let(x,true) | Rtimer(t) >x> let(x,false) )
```

Non-strict Evaluation; Parallel-or

Parallel-or is a classic problem in non-strict evaluation: computation of $x \vee y$ over booleans x and y publishes *true* if either variable value is *true*; therefore, the evaluation may terminate even when one of the variable values is unknown. Here, we state the problem in Orc terms, and give a simple solution.

Suppose sites M and N publish booleans. Compute the parallel-or of the two booleans, i.e., (in a non-strict fashion) publish **true** as soon as either site returns **true** and **false** only if both sites return **false**. In the following solution, site

`or(x,y)` returns $x \vee y$. Site `if(b)` returns `true` if `b` is true; it does not respond otherwise.

```
let(z) where z in
  (if(x) | if(y) | or(x,y)
   where x in M
   where y in N)
```

Sequences

Orc does not have a primitive notion of structured data. However, it is easy to use sites to simulate many structured data needs. For example, consider a site that represents a set of items; each call to the site publishes one value from that set that it has not previously published. When there are no more items left in the set, calls to the site remain silent. We call such a site a *sequence*.

We can define a recursive expression `forall` that will publish all of the values in the sequence:

```
def forall(S) = S >v> ( let(v) | forall(S) )
```

Notice that this definition of sequence allows us to easily interface with unbounded data streams. For example, we can take a site `listener`, which listens for incoming network traffic, treat it as a sequence, and stream its publications to some other expression as they arrive.

```
forall(listener) >packet> Process(packet)
```

Sometimes, we will need to iterate over a sequence, perform an operation on each item, and wait for all such operations to complete before proceeding. This is a generalization of the fork-join example shown earlier. This operation only makes sense for finite sequences, so we assume the existence of an additional site

`hasmore(S)` which returns `true` if the sequence `S` has more items, or `false` otherwise. Note that sites and expressions are first-class values in Orc, so we can take the operation `oper` as a parameter and then invoke it on a data item.

```
def foreach(S, oper) =
  if (more) >>
    S >item>
      ( let(this, rest)
        where this in oper(item);
          rest in foreach(S, oper) )
  | if (~more) >>
    Signal
  where more in hasmore(S)
```

2.2 XQuery

XQuery is a declarative query language that provides full language support for manipulating and querying XML data. It leverages XPath 2.0 expressions to efficiently navigate XML's tree-structured data. In our implementation, we use XQuery 1.0 with update semantics [6] to support mutable XML data. Although we have support for the complete XQuery 1.0 specification, we intentionally avoid using FLWOR expressions in our implementation of the Narada protocol. FLWOR is traditionally used for iterating over XML data. In our Narada example in Chapter 4, iterators are constructed with sequence sites.

In the following example, the XQuery expression uses XPath to select the address of each neighbor in the local neighbor table, which is defined below.

```
$data/neighbors/neighbor/@address
```

Assuming that the data is in the following XML format:

```
<data>
```

```
<neighbors>
  <neighbor address="192.168.1.1"/>
  <neighbor address="192.168.1.2"/>
</neighbors>
</data>
```

The XPath expression `$data/neighbors/neighbor` selects all of the neighbor elements from `neighbors` (attributes are denoted with `@`) and returns the following list of entries:

```
<neighbor address="192.168.1.1"/> <neighbor address="192.168.1.2"/>
```

2.3 Narada

Narada is a distributed, asynchronous, self-organizing protocol that manages the structure of an ad-hoc, peer-to-peer network. Popular file-sharing applications like Gnutella and Kazaa operate in this type of environment. Infrastructure management protocols like Narada are necessary for maintaining the integrity and performance of such systems, including routing and end-system multicast [7]. Narada's management capabilities include organizing nodes into an *overlay* network, which sits on top of a logically-linked (i.e. a physically connected) mesh network of nodes.

Narada manages small groups of participants (i.e. *end systems*) which adapt to changes to the group structure. In the protocol, network latencies determine the immediate connections to neighbor nodes. In order to ensure that neighbor latency information is current, each peer in the overlay network periodically pings its immediate neighbors. For example, if a node sends a ping message to a new neighbor then the address of the new neighbor is added to the calling node's local neighbor table. Additionally, routing messages are sent to peers in the member group to determine current latencies and shortest paths. The routing algorithm ensures that relatively long latency paths are replaced by shorter more direct paths, which

is essential to maintaining fast network multicast and unicast. Finally, the Narada protocol requires a peer to periodically send refresh messages to its neighbors, which ensures that all neighbors have the latest information about the peer's status in the overlay network. By sending remote messages and sharing the local member table with each node's neighbors, each peer gets a greater view of the network.

In summary, we presented an introduction to the Orc programming language through a simple weather forecast example. We also showed examples of how Orc encodes common concurrent programming patterns such as fork-join parallelism in addition to time-out patterns, non strict evaluation and sequences. Orc expresses and orchestrates distributed computations in an elegant manner. Finally we presented a high-level overview of the XQuery language and the Narada protocol. In Chapter 4 we discuss the Narada implementation in greater detail.

Chapter 3

Orc-X language overview

The goal of Orc-X is to bring an appropriate data model to the Orc language and to demonstrate how well they work together in expressing the semantic needs of distributed applications like Narada. The Orc-X language combines Orc with XML and XQuery. Orc is the *master language*, which manages the orchestration of concurrent and serial computations. XQuery expressions are embedded inside of Orc expressions to manage the data in our XML data model. In this section, we discuss the syntax, semantics and implementation of Orc-X as an extension of Orc. Orc-X is a prototype for a general approach to embedding other data models and languages within Orc. It is conceivable that other data models (e.g. SQL-like relations) will be embedded in Orc to suit the needs of other application domains.

3.1 Syntax and semantics

As demonstrated in the examples that follow, Orc and XQuery differ syntactically. Rather than add the syntactic elements of XQuery directly into the Orc grammar, which would complicate parsing and would not represent a general solution to embedding languages in Orc, we instead isolate XQuery syntax from Orc syntax. Orc-X

$f, g \in Expression$	$::=$	$\mathbf{0}$	Zero expression
		$ M(P)$	Site call
		$ E(P)$	Expression call
		$ f \mid g$	Symmetric Parallel Composition
		$ f > x > g$	Sequential Composition
		$ f \mathbf{where} x : \in g$	Asymmetric Parallel Composition
		$ \{e\}$	XQuery expression
$e \in XQueryExpression$			

Figure 3.1: Orc-X extension of Orc syntax

introduces a new base expression $\{e\}$ into the syntax of Orc (Figure 3.1), where e is an XQuery expression. e is *open*; it may contain external variables that are not defined within the expression itself. Those external variables correspond to Orc variables in the scope surrounding the expression.

An embedded XQuery expression may run when all of its external variables have been bound; like sites, embedded expressions are *strict*. It executes the corresponding XQuery code, substituting the values of the bound variables into the query. An XQuery evaluation completes by publishing a *sequence*, as defined in the XPath data model [11]. If the sequence is empty, the expression remains silent. We use the techniques shown in Chapter 2 to manipulate sequences in Orc.

XQuery’s representation in Orc

Rather than extending the core semantics of Orc to handle XQuery evaluation, we instead express executions of XQuery expressions by encoding them as site calls.

For each embedded expression $\{e\}$, we define a new site call $M(\bar{x})$, where the variable names \bar{x} used in Orc are the same names that appear in the XQuery code. The site M executes the operations corresponding to the query. The arguments to M are the values which will be substituted for each free variable in that expression.

For example, suppose we are writing an Orc program to examine a bug

database written in XML, referenced by the Orc variable `tracker`. We want to query the XML data for any bug records whose priority exceeds `n`, another Orc variable. We write the Orc-X expression:

```
{ $tracker/bugs/[@priority > $n] } >S> Report(S)
```

This encodes into Orc as an invocation of a site `M`, where $M(v,w)$ evaluates the XQuery expression `v/bugs/[@priority > w]`. This encoding results in the Orc expression:

```
M(tracker, n) >S> Report(S)
```

This encoding strategy gives Orc-X the full expressive power of XQuery with no changes to the original semantics of Orc.

3.2 Orc-X implementation

Figure 3.2 presents a high-level view of our current implementation of embedding XQuery into Orc. The Orc language is implemented in Java and leverages the Java-based Galax API for evaluating XQuery expressions. There are presently two phases required to integrate XQuery expressions with Orc expressions: parsing and evaluation. For the parsing phase, each Orc-X peer interfaces with a running instance of the Galax [12] XQuery server through HTTP to parse the initial query and return a list of external variables. For the evaluation phase, Orc executes functions in the Galax API to perform the evaluation of the query in the context of the enclosing scope. We anticipate in future work to extend the Galax API to support parsing as a single step, thereby obviating the need for interfacing with the Galax query server for parsing.

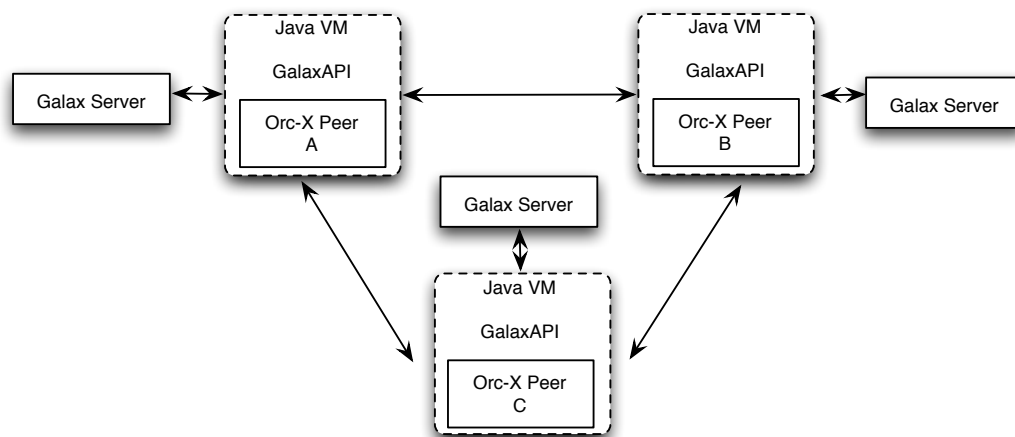


Figure 3.2: Orc-X architecture: All inter-peer/server communication is through HTTP

Parsing

An Orc-X program is first processed by the local Orc engine, parsing the Orc code and delegating the parsing of the embedded XQuery code to the Galax query engine. We modified the ANTLR parser to recognize `{ }` as the boundary between Orc code and XQuery code. Since XQuery is embedded inside of Orc, the parsing phase pauses when it encounters a `{`, serializes the XQuery within the braces and sends it as an HTTP message to the local Galax server. After parsing the query, the Galax server returns any errors encountered and a list of external variables, which correspond to bound Orc variables in the enclosing scope.

Evaluating XQuery expressions

Each XQuery expression is evaluated through the Galax API with the evaluation context from Orc (also includes standard XQuery libraries). Again, the query is parsed, but at this point it is type-checked (all external variables have bindings). The result is stored in the Java VM space and is accessible by the local Orc peer,

which is running in the same VM space.

Network primitives

In order to build complex, distributed applications like Narada, we implemented a simple client-server protocol based on HTTP that permits communication between Orc-X peers. We assume the existence of a small set of sites for managing network capabilities. This allows us to abstract away particular network details, and makes simulations easier. Each node on a network has a string describing its address; these addresses are used as parameters to the network sites. The three network sites we assume are as follows:

`Ping(addr)` publishes the network round-trip time from the current machine to the machine with address `addr`; if that machine is down or unavailable, this site remains silent.

`Send(addr,msg)` serializes the data item `msg` and sends it to the machine with address `addr`. The data can be of any type; in practice, our implementation example uses XML messages. `Send` is asynchronous, so it publishes a signal as soon as the data is sent, regardless of when or whether it is received.

`Receive` publishes any one waiting data item that has been sent to this machine.

Extending Orc with XML data makes network applications easier to write, since it is easier to construct, serialize, and examine complex messages. These simple network capabilities, together with the XML handling capabilities of Orc-X, are all that we need in order to implement complex network applications, such as the Narada protocol. Throughout the implementation of the Narada protocol, nodes send information in their local tables to other nodes; they do so by sending XML messages containing this information using these network operations, and listening for incoming XML messages from other machines.

New Orc sites

In addition to the three network primitives discussed above (i.e. `Ping(addr)`, `Send(addr,data)` and `Receive`) the other fundamental sites added to the Orc implementation are as follows:

1. `Signal` is a type of control flow data that indicates when an expression has terminated or published a value.
2. `hasmore(S)` examines each `GalaxValue` in `GalaxListValue` of sequence `S` and returns true if `S` has more values remaining, false otherwise. Used exclusively in `foreach`, where we iterate over values. We discuss `GalaxValue` and `GalaxListValue` later in this chapter.
3. `empty(S)` determines if the sequence `S` is empty.
4. `print(msg)` prints out a string debug message `msg`.
5. `gettime()` returns the current local time in seconds.
6. `boolValue({XQuery_expr})` converts a `GalaxValue` primitive boolean into a native Java boolean value.
7. `GuiReport(origin,target,gui_msg_type,msg_type,msg_content)` sends message traffic information via HTTP to GUI for logging, including the origin of the message, target destination, the type of graphical message, the name of the message and the content of the message.
8. `GuiStatus(origin,graph_type,graph)` sends updates via HTTP to the GUI containing the new graph structure (e.g. connections between nodes) encoded in `graph`.
9. `GuiNodeDie` immediately kills a specified node that is currently alive. Forces a `System.exit(-1)` on the node's running instance of the Orc engine.

10. `Lock` creates a "lock" for a critical section of code. The implementation does not use Java locks, but rather simulates a lock. We discuss this in greater detail in later in this chapter.
11. `Unlock` unlocks the "lock" once a critical section completes execution.

Data management

All data management is handled by Galax, including XML documents and all basic scalar data types such as integer, string, double, etc. While the implementation of Orc supports its own data model with basic scalar types, Orc-X maintains data model consistency by relying on Galax to manage all data. Only the data types necessary for Orc-X to manage control flow are supported, including `Signal`, boolean values for use in `if` and integer values for use in `Rtimer`.

The Galax Java API provides access to Galax-managed data through opaque handles represented as integer unique identifiers (UIDs). A top-level Java class called `Item` represents all Galax datatypes, including basic scalars and complex nodes such as XML documents. The type `ItemList` represents a sequential list of `Item` values; Galax makes no distinction between an `Item` and an `ItemList` containing only one value. The Orc engine wraps values of type `Item` and `ItemList` inside `GalaxValue` and `GalaxListValue`, respectively, in order to provide common accessors to the underlying Galax data value and type information. The opaque handles provided by Galax act as references, allowing their subsequent use in additional calls to the Galax API.

Predominantly, `GalaxValue` and `GalaxListValue` serve as couriers for the underlying Galax values; aside from debugging support there is often no need to examine the Galax values within Java. However some of the primitive Orc-X sites such as `Ping` and `Send` must unwrap their parameters from Galax into native Java types for use in Java communication APIs. Both `GalaxValue` and `GalaxListValue`

provide a `coreValue()` method for extracting a Java `String` representation of the Galax value. The `itemKind()` method in the Galax API provides insight into the underlying `Item` type, and additional API methods allow the Orc engine to extract a usable string representation for types such as `Attribute`, `Element`, and so on. For entire XML documents the Orc engine uses `serializeToString()`.

Galax Java API and Galax engine modifications

We had originally anticipated interfacing Orc with the eXist XQuery engine [17], but found their API to be cumbersome and not fully compliant with the standard XQuery update semantics defined in [6]. Instead, we chose to interface with the Galax XQuery implementation. Galax is a fully-compliant, open-source implementation of XQuery with a Java-based API. While the API was fairly complete, we were able to make some contributions. For example, we extended the API to include a key method in `ProcessingContext` called `setLanguageKind("dxq")`. This method allows Orc to pass additional information to Galax to configure Galax to accept DXQ language syntax (i.e., this option turns on all of the XQuery update semantics).

The Galax Java API contains several layers that interface to the XQuery engine. The XQuery engine is written in the OCaml language. OCaml provides a C interface which Java connects to through the Java Native Interface (JNI). The process of extending the Java API requires modifications in three different languages, which proved to be quite challenging. While the Galax API does not support reentrant execution, Orc's concurrency model is supported by cooperative multitasking in a single thread such that no concurrent accesses to the API will ever occur.

We modified `galax_server.ml` in the Galax engine to support checking free variables from an incoming HTTP message containing an XML document or XQuery. Upon receipt of an incoming message from the Orc engine, Galax parses

the message and collects the payload (i.e. the XQuery/XML data) and determines which variables are "free", i.e. externally defined within the scope of Orc. It returns a message back to Orc via HTTP containing any errors encountered in parsing the XQuery expression and a list of externally-defined variables (which are bound later within the enclosing scope of the Orc expression).

Locking

While individual Galax API calls are atomic, Orc-X programs may require atomicity for a more granular read-modify-write sequence of XQuery operations. To prevent collisions with concurrently running XQuery operations that access the same Galax data objects, Orc-X introduces fundamental sites called `Lock` and `Unlock`. The Orc engine uses a queue of ready tokens that represent units of execution ready for processing in the cooperative multitasking model. Our implementation of locks adds a single boolean for the lock state, along with an additional queue of pending tokens that are waiting on the lock. If the lock is not yet acquired - i.e. its state is `false` - then a call to `Lock` will change the lock state to `true` and continue adding tokens to the *ready queue* as normal. When a lock is already acquired, additional calls to `Lock` will result in the subsequent tokens being placed on the *pending queue* instead of the *ready queue*. A call to `Unlock` will remove the first token from the *pending queue* and place it on the *ready queue*, allowing the start of a new critical section to proceed. When no tokens remain on the *pending queue*, a call to `Unlock` will set the lock state back to `false`.

Visual debugging via Adobe Flash Player GUI

In our previous work in [10] we added the capability of visually debugging a running simulation of the Narada protocol on several Galax engines through a Flash Player GUI. For Orc-X we extended this GUI to debug the Narada protocol running on

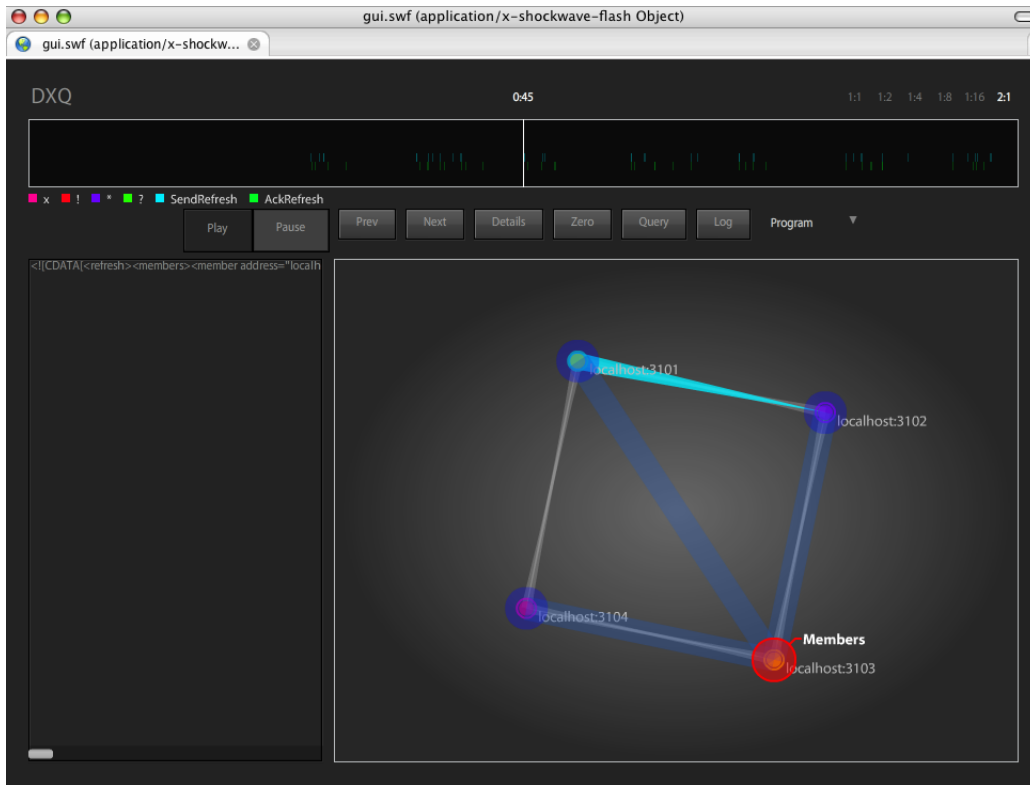


Figure 3.3: Flash Player GUI with four Orc engines running Narada

several Orc engines. The Flash Player listens to incoming HTTP requests containing XML data that is sent from each running Orc engine. The XML information includes the message traffic between the Orc-X nodes as well as updates to the visual display of the status of the nodes. The GUI visually tracks all message traffic and displays any changes to the overlay graph of running nodes. By having a visual representation of the concurrent computations, it can help elucidate the causes of unexpected behaviors in the distributed system. See Figure 3.2 for a snapshot of the GUI in action.

Debugging trace of Galax API calls

We added a tracing mechanism to the Orc engine that tracks every Galax API operation executed (e.g. query evaluations, construction of new Atomic types, ItemLists, etc.). After executing a sample OrcX program, it produces a `trace.log` file that can be played back in a separate, simpler test harness to reproduce the identical sequence of Galax API calls. The trace captures each operation along with the original UIDs associated with each query/XML value from the OrcX program. As each operation is played back in the test harness the result is associated with the original UID using a HashMap, which allows further operations to reference the new objects via the original UID. This allows playback of all API operations seen from a given OrcX node. We used this tracing mechanism to isolate API-specific problems, of which there were many including some serious memory bugs in the C part of the API.

Chapter 4

Experimental evaluation

4.1 Narada implementation in Orc-X

Narada is a distributed, self-organizing protocol that monitors and reacts to changes to the group membership of the participating end systems in the overlay network. The end systems enter the group with limited knowledge of the physical topology. Our implementation of the Narada protocol starts up as follows:

```
{<data><self>localhost:SELF</self>
  </data>} >data>
{<neighbors><neighbor>localhost:N1</neighbor>
  <neighbor>localhost:N2</neighbor>
</neighbors>} >neighborSet>
initialization(data, neighborSet) >> ...
```

`data` is a reference to a fresh XML document for this node's state, initially containing only one item, `self`. Since we simulate the protocol locally and map each node to a port address, `SELF` is the designated port for this node (a valid value for `SELF` would be, for example, 3101). Then, after binding the XML document to `data`, we pass it to `initialization(data, neighborSet)`. The second value

passed to `initialization` is a sequence of addresses for the immediate neighbors of this node. This value is bound to the variable `neighborSet`.

Core protocol

Continuing where we left off from above, the core expression defining the entire protocol is simply:

```
initialization(data, neighborSet) >>
  ( listen_for_messages(data)
    | routing_metronome(data)
    | refresh_metronome(data)
    | latency_metronome(data)
    | neighbor_metronome(data))
```

The `initialization` expression prepares the local `data` and `neighborSet` for use in the protocol, and publishes a signal only when this preparation is complete. Then, in parallel, the node begins to `listen_for_messages` from other nodes (see definition on page 32), and also begins to send four different types of protocol messages at regular intervals.

Initialization

The algorithm begins by installing initial values in the empty XML document given by `data`. The initial member and routing tables are empty except for a self entry for this node's own address. An iteration over the `neighborSet` populates the initial neighbors table, using a helper function `addNeighbor`. Notice in the code below that we use the fork-join idiom to represent the fact that neighbor and member initializations are independent of each other and may occur in parallel. The power of inlined XML is immediately apparent here; we can use Orc's combinators to

manage iteration, dataflow, and joins, while writing XML that has Orc variables embedded directly into it.

```
def initialization(data, neighborSet) =
  let(init-routes) >>
    let(init-neighbors, init-members) >>
      Signal
      where init-routes in {insert <routes> <route address="{self}"
                                distance="0"
                                latency="0"/>
                                </routes> into $data};
      init-neighbors in ({insert <neighbors> </neighbors> into
                          $data} >>
                          foreach(neighborSet, addNeighbor));
      init-members in {insert <members>
                            <member address={self}>
                              <sequence> "0" </sequence>
                              <live>"1"</live>
                            </member>
                          </members> into $data}
```

Probes

The core of the Narada protocol is a small set of periodic, asynchronous probes of immediate neighbors. Since the end systems that join the network have a limited understanding of physical topology, these probes are used to figure out for example, latencies to other participating end systems. There are four such probe actions:

1. A routing probe sends updated routing information (e.g. latencies and hop counts) to a neighbor.

2. A refresh probe broadcasts the contents of this node's member table to each of its neighbors, ensuring that those neighbors have up-to-date information about the overlay.
3. A latency probe recalculates the expected round-trip time to contact a randomly chosen member of the overlay network (not necessarily a neighbor).
4. A neighbor probe checks the timestamp of our last contact with a neighbor. If the elapsed time since our last contact exceeds a threshold, that neighbor is marked inactive.

We use `Metronome` to initiate these probes at regular time intervals. The expression `Neighbors` uses an XQuery to find the address of each neighbor in the local neighbor table, so that we can probe each neighbor individually.

```
def Metronome(t) = Rtimer(t) >> (Signal | Metronome(t))

def Neighbors(data) =
  forall({$data/neighbors/neighbor})

def routing_metronome(data) =
  Metronome(5000) >> Neighbors(data) >n>
  routing_probe({$n/@address},data)

def refresh_metronome(data) =
  Metronome(3000) >> stepseq(data) >> Neighbors(data) >n>
  refresh_probe({$n/@address},data)

def latency_metronome(data) =
  Metronome(7000) >> random({$data/members/member}) >m>
  if(~empty(m)) >>
```

```
latency_probe(m,data)
```

```
def neighbor_metronome(data) =  
  Metronome(9000) >> Neighbors(data) >n>  
  neighbor_probe(n,data)
```

We show implementation details for all four probes.

Routing probe: broadcasting local routing data

We implement `routing_probe` by using the fundamental `Send` site mentioned earlier to send messages to peers on the network. Embedded XQuery expressions allow us to concisely construct structured XML messages to send across the network, shipping our local routing data to each neighbor.

```
def send_message(addr, content, data) =  
  Send(addr, msg)  
  where msg in  
    {<message origin="{self}">  
      { $content }  
    </message>}  
  
def routing_probe(n,data) =  
  send_message(n, content,data)  
  where content in  
    {<routing>  
      {$data/routes}  
    </routing>})
```

Refresh probe: broadcasting local member data

`refresh_probe` is implemented very similarly to `routing_probe`, in which a message containing local data is sent to all neighbors. The only difference is in the content of the data; we instead ship member data instead of routing data. For each end system, Narada creates two overlay spanning trees of the CVG: one that keeps track of shortest paths and a second that tracks group membership. The former is created as a result of the routing probe messages and the latter is generated by the refresh probes.

```
def refresh_probe(n,data) =
  send_message(n, content,data)
  where content in
    {<refresh>
      {$data/members}
    </refresh>}}
```

Latency probe: checking response times

The `latency_probe` refreshes our local data about the round-trip time to another member of the network. Notice that the computation of `latency` takes advantage of Orc's simple encoding of a time-out strategy. Using a **where** clause, we establish a time limit on the call to `Ping`, and remove that member from our member set if the ping does not respond by the time limit.

```
def latency_probe(m,data) =
  if(boolValue({$latency < $limit})) >>
    setLatency(m, latency)
  | if(boolValue({$latency >= $limit})) >>
    removeMember(m)
  where latency in (Ping(addr) | Rtimer(orclimit) ) >> {$limit};
  limit in {100};
```

```

addr in {$m/@address};
orcLimit in let(1000)

```

Neighbor probe: checking neighbor liveness

The `neighbor_probe` decides if a neighbor is still alive by checking its local record of the last time its neighbor had sent out a refresh message. If the timestamp, i.e. the amount of time that has elapsed since the node received a refresh message from its neighbor, is greater than some arbitrarily-specified threshold then the neighbor is removed from the node's local neighbor table and marked inactive in the local member table. Note that in the case where no action is needed (i.e. `~doRemove`), we signal immediately so that the caller of `neighbor_probe` may proceed with other actions such as reporting to the GUI.

```

def neighbor_probe(n,data) =
  forall({$data/members/member[$n/@address = @address]}) >m>
    if(doRemove) >>
      removeNeighbor(addr,data)
  | if(~doRemove) >>
    Signal
  where doRemove in boolValue({$dt > $thold});
        dt in gettime() >current> {$current - $m/time};
        addr in {$n/@address};
        thold in {30}

```

Message listeners

In addition to sending messages to neighboring nodes, a node participating in the protocol must also listen for updates from its neighbors and incorporate those updates into its own state. Here, we use the `Receive` site as a sequence value site, to

continuously listen for network traffic. Embedded XQuery expressions examine the content of XML messages sent by neighbors, and call our defined handlers to update the local state at this node. Thus, with XML as a data model, we can quickly implement RPC-like functionality with very few assumptions about the capabilities of the underlying network.

```
def listen_for_messages(data) =
  forall(Receive) >msg>
  {$msg/@origin} >origin>
  if(~empty(origin)) >>
    ( {$msg/refresh/members} >m>
      if(~empty(m)) >>
        refresh_handler(m,data)
    | {$msg/routing/routes} >r>
      if(~empty(r)) >>
        routing_handler(origin,r,data)
    | Lock() >>
      addNeighbor(origin,data) >>
      Unlock()
    )
```

There is a synchronization concern here: the `forall(Receive)` expression will publish any time a message is received, potentially causing multiple instantiations of the proceeding expressions. The expression `addNeighbor` updates local state so a lock is necessary to prevent collisions from simultaneous invocations. `refresh_handler` and `routing_handler` have their own locks as needed within their definitions, which we discuss below.

Message handlers

`refresh_handler` uses a `forall` expression to iterate over the members in the neighbor's (i.e. remote) member table that was received in `listen_for_messages`. Locking is done within each iteration. Due to the use of `forall`, `refresh_handler` will never signal when it has completed. However it is necessary to have a complement for every `if` expression to ensure that all paths signal to the subsequent `Unlock` site call.

```
def refresh_handler(neighbor_members,data) =
  forall({$neighbor_members/member}) >r_m>
    {$r_m/@address} >addr>
    Lock() >>
    ( if(~doProcess) >>
      Signal
    | if(doProcess) >>
      ( if(doInsert) >>
        insertMember(r_m,data)
      | if(~doInsert) >>
        updateMember(l_m, r_m)
      where doInsert in empty(l_m);
        l_m in {$data/members/member[@address=$addr]}
      )
    where doProcess in boolValue({$data/self != $addr})
    ) >>
  Unlock()
```

`routing_handler` has a similar structure to `refresh_handler`, however it uses XQuery update semantics directly instead of using helper functions. The call to `Lock` protects the read/update critical section and ensures that only a single route is stored for a given address. The code tests for an existing route and either inserts

or replaces as appropriate, which cannot be exposed to concurrent updates to the routing table. Note again that all control-flow paths within the critical section must signal to ensure that Unlock is invoked.

```

def routing_handler(neighbor,neighbor_routes,data) =
  setup(neighbor, data) >latency>
  forall({$neighbor_routes/route}) >nr>
    if(~empty(nr)) >>
      if(empty({$nr/hop[(@to|@from)=$data/self]})) >>
        Lock() >>
          ( {$data/routes/route[@address = $nr/@address]} >Routes>
            ( if(doInsert) >>
              {insert node $new_route into $data/routes}
            | if(~doInsert) >>
              ( if(doReplace) >>
                {replace node $Routes with $new_route}
              | if(~doReplace) >>
                Signal
              )
            where doReplace in
              boolValue({$nr/@distance + 1 < $Routes/@distance})
          )
        where doInsert in empty(Routes);
        new_route in
          {$nr/hop[1]} >fst_hop>
          {attribute to {$fst_hop/@from}} >to_attr>
          {<route address="{ $nr/@address}"
            distance="{ $nr/@distance + 1}"
            latency="{ $latency + $nr/@latency}">
            {(<hop from="{ $neighbor}">{$to_attr}</hop>, $nr/hop)}
          </route>}

```

```
) >> Unlock()
```

The full implementation of the Narada protocol in Orc-X is available in Appendix A of this thesis.

4.2 Evaluation

We tested Orc-X and our implementation of the Narada protocol by simulating Narada on an 8-core Intel Xeon computer – each core running at 2.6GHz. Our evaluation focused on three areas detailed in the following subsections: Correctness, Performance and Expressiveness.

We compared ourselves to DXQ for several reasons. First, DXQ has a publicly-available, full implementation of Narada. Also, Orc-X and DXQ use the same underlying query engine infrastructure (Galax [12]). Aside from correctness and performance comparisons, these factors made the two implementations more relatable in a code-expressiveness comparison.

Correctness

We tested several network configurations consisting of 3 to 7 simulated nodes, where each node was running a separate instance of the Orc engine. We examined the behavior of our implementation of Narada and compared it to DXQ’s implementation of Narada, which was simulated on the same computer. Both implementations exhibited the same behavior: achieving stability after a short period of time, whereby no subsequent state changes occur at each node.

We extended DXQ’s logging and visualization mechanism for Orc-X, which allows us to record and visually display the inter-peer message traffic and changes to node state. Thus, we can capture movies of our simulations and visually compare the two implementations to see if the two languages yield the same network config-

uration. Figures 4.1 through 4.5 are screen captures of some interesting points in our Orc-X simulations with 5 nodes that demonstrate the correctness of our Narada implementation. For brevity, we don't include the same screenshots from our DXQ simulations with 5 nodes, but it should be noted that the DXQ simulations yielded the same configurations as depicted in our figures below.

Our simulation began with the 5 nodes entering the GUI completely unconnected. Then, `initialization` is invoked and constructs a connected graph, or *mesh*, as depicted in the bottom screenshot in Figure 4.1 (see page 44). Following initialization, the 4 probes execute simultaneously on all 5 nodes and start updating the members and routing overlay trees, depicted in Figure 4.2 and 4.3. The top screenshot in Figure 4.2 depicts the members overlay tree after a few seconds of execution and the bottom one demonstrates after about 30 seconds that node 3102 has detected all of the members in the mesh. Member steady state is reached once all of the nodes detect all of the *alive* members in the mesh, which is reflected in the member overlay tree for each node. Similarly, routing steady state is achieved once all of the nodes detect all of the alive members in the mesh. In Figure 4.3, the bottom screenshot depicts the routing steady state for node 3102.

Figure 4.4 depicts a scenario where we explicitly killed node 3105. The top screenshot shows that node 3102 has detected that 3105 is no longer functioning and hence, removed it from its member overlay tree (and subsequently marked it as "dead" in its local state). Additionally, we see that 3102 has removed node 3105 from the routing overlay tree. It should be noted that all of the nodes were able to detect and update their respective overlay trees, and hence achieved a steady state after the change to the mesh. Around 22 seconds into execution of this scenario, we killed node 3105, and around 52 seconds we saw the changes starting to be reflected in the overlay trees for the other nodes. This makes sense because in our implementation of Narada, `neighbor_probe` removes members that haven't sent out

refresh messages within a 30 second threshold.

Our final scenario depicted in Figure 4.5 demonstrates the correctness of our implementation of the shortest paths routing algorithm. We loaded a pre-configured latency table into our configuration for this simulation, and used these static, unit-less latencies instead of invoking the `Ping` method as we had done in previous simulations. As a result, we were able to deterministically test the correctness of the routing code. The configuration consisted of weighting the immediate link between node 3102 and 3101 with a latency of 100, while all other immediate neighbor links were weighted at 10 each. Therefore the path, 3102-3103-3104-3105-3101, is the desired route between nodes 3102 and 3101 since the latency through this path (40) is better than the direct link.

Performance

We tried various Java profiling tools, but unfortunately due to difficulties with handling JNI calls we were only able to obtain results from HPROF [20]. We collected results pertaining to the CPU usage of our system. Our system consisted of four nodes simulating the Narada protocol in the Orc-X language. Results were accumulated after processing 700 XQuery expressions. Table 4.1 shows the results obtained through CPU sampling. The table lists the top ten most frequently executed methods. The majority of the time was spent collectively in `socketAccept` and in the evaluation method in the Galax API. `socketAccept` is used in the underlying TCP connection, where Orc engines interface with each other by sending HTTP messages over this connection. In our configuration of the Narada protocol, each node sends a refresh message every 3 seconds, a routing message every 5 seconds and a latency ping message every 7 seconds. Since the Narada protocol continually sends periodic messages around the mesh network, it is not too surprising that a large portion of the system's time is spent on network connectivity.

The results collected in Table 4.2 reflect the HPROF CPU time (i.e. wall clock time) of the same four-node system simulating Narada in Orc-X. The table lists the top ten most time-intensive methods. Time-based profiling does not accurately identify performance bottlenecks in a system that has idle periods of waiting, for example due to the use of sleep timers or locks. In the case of Narada in Orc-X, we see that four of the top five methods represent idle activity that accounts for 60% of wall-clock execution time. While Table 4.1 shows *galapi.Galax.nativeEvalStatement* as representing 27% of CPU utilization, Table 4.2 shows that the same method only consumes 6.5% of wall-clock time.

Table 4.1: HPROF CPU samples from 4 nodes running Narada

rank	self	accum	count	method
1	63.72%	63.72%	1728	java.net.PlainSocketImpl.socketAccept
2	27.06%	90.78%	734	galapi.Galax.nativeEvalStatement
3	4.87%	95.65%	132	java.net.SocketInputStream.socketRead0
4	0.77%	96.42%	21	galapi.Galax.nativeLoadStandardLibrary
5	0.44%	96.87%	12	galapi.Galax.nativeEvalStatement
6	0.29%	97.16%	8	java.io.FileOutputStream.writeBytes
7	0.29%	97.46%	8	galapi.ItemList.nativeSerialize
8	0.15%	97.60%	4	galapi.Galax.nativeInitialize
9	0.11%	97.71%	3	galapi.Galax.nativeEvalProgram
10	0.07%	97.79%	2	java.lang.ClassLoader.defineClass1

Performance improvements to Orc-X system

The discrepancies between time-based profiling and sample-based profiling indicate that the execution of Orc-X has bursts of execution in between idle periods. This correlates well to Narada’s use of probes and message handlers, which alternate between idle and active. To improve the performance of active CPU utilization we targeted the use of Galax API routines such as *nativeEvalStatement* and *loadStandardLibrary*.

Table 4.2: HPROF CPU time (in ms) from 4 nodes running Narada

rank	self	accum	count	method
1	38.10%	38.10%	22	java.net.PlainSocketImpl.accept
2	16.79%	54.89%	9	java.util.concurrent.locks.LockSupport.park
3	6.47%	61.36%	561	galapi.Galax.evalStatement
4	3.37%	64.72%	3	java.lang.Object.wait
5	1.80%	66.52%	2	java.lang.Object.wait
6	1.06%	67.58%	137	java.net.SocketInputStream.read
7	0.95%	68.54%	38741	sun.nio.cs.ASCIIEncoder.encodeArrayLoop
8	0.46%	69.00%	22	galapi.Galax.loadStandardLibrary
9	0.43%	69.42%	78050	java.util.regex.PatternBmpCharProperty
10	0.26%	69.69%	38741	sun.nio.cs.ASCIIEncoder.encodeLoop

The data model integration for Orc-X requires all values (including constants) to be evaluated through the Galax API. Constant XQuery expressions are a special case and are known to the Orc engine because they have no external variables. To eliminate redundant evaluation of these constant expressions we use a cache to store the results of the first evaluation of each expression. Constant expressions such as {"g"} and {30} are common in our Narada implementation, and Table 4.3 shows that constant-query caching has a hit rate of 20%. While this results in a significant reduction in calls to *galapi.Galax.nativeEvalStatement*, the performance gain is limited by the fact that these are the simplest queries for Galax to perform. Table 4.4 shows the results of the performance gains from query caching and prolog caching, discussed below. We collected timing numbers by manually instrumenting the code to accumulate measures of wall-clock time immediately before and after calls to the Galax API to evaluate an XQuery.

We removed additional overhead by caching the Galax data structures required for expressing external variables. Evaluating an expression using Galax requires a preliminary step to establish and load the *main module* along with any external variable declarations. This is known as the *module prolog*, and its overhead

is unique to each combination of external variables that must be declared. However expressions which are executed repeatedly – or those which share a common set of external variables with other expressions – can reuse an existing module prolog that was constructed with the same external variable declarations. This prolog caching eliminates some Galax API overhead, and despite compulsory misses Table 4.3 demonstrates that it approaches a 100% cache hit rate. Table 4.4 shows that prolog caching provides better performance gains than constant query caching

Table 4.3: Cache hit rates for 4 nodes running Narada

	Node 1	Node 2	Node 3	Node 4
Query cache hit rate	20.200%	20.000%	20.500%	20.300%
Prolog cache hit rate	97.243%	97.250%	97.233%	97.240%

Table 4.4: Execution time (in ms) for 4 nodes running Narada

	Node 1	Node 2	Node 3	Node 4
No caching	11053	11120	11174	11197
Query caching	10891	10709	10694	10816
Prolog caching	10713	10607	10680	10648
Both caching	10210	10466	10487	10491
% Caching speedup	8.3%	6.2%	6.5%	6.7%

Performance: scalability of Orc-X implementation

It is well-known that language integration through APIs is inefficient and doesn't scale as well as native integration. The main goal in creating Orc-X was to bring a suitable data model to the Orc language to build complex, distributed applications like Narada. In our implementation work, we focused on achieving this goal by a quick-and-dirty solution: integrating the Java-based Orc engine with XQuery/XML

through the Galax Java API. The added layers of abstraction allowed us to complete the integration more quickly, but at the cost of overall system complexity; the Galax API spans three different languages, using C to bridge between Java and OCaml.

With regard to scalability, we discovered that the Galax API has memory management issues in the C layer which limit the long-term stability of our system. We are the heaviest users of the API, and our trace playback debugging tool described in Chapter 3 helped the Galax community isolate and work around the most serious bugs. In particular, a severe discrepancy between Java and C over the presumed lifetime of certain objects was causing premature memory deallocation. However the workaround of never deallocating these objects is likely a source of memory leaks. We suggest that a truly scalable, Java-based Orc-X implementation should contain a native integration of XQuery in Java.

Expressiveness

As a process calculus, Orc is able to express concurrency in a more succinct manner than traditional, Algol-like languages. Even declarative, distributed languages such as DXQ have trouble expressing concurrency with fine granularity. Below we discuss the expressiveness of Orc-X by comparing our Narada implementation to the same implementation in DXQ.

Expressiveness: Narada in DXQ vs. Orc-X

Ignoring whitespace, debugging print statements and calls to the GUI, there are 305 lines of DXQ code that implement the Narada protocol. Of these lines, 38 are spent calling lock/unlock. In comparison, there are 229 lines of Orc-X code (also ignoring whitespace) in the Narada protocol implementation. This count includes 12 calls to lock/unlock. Because of the clear syntactic separation between Orc and the XQuery data model, we can observe that 108 lines (47%) include XQuery/XML code. With

XQuery as the common ground between Orc-X and DXQ, we can deduce that Orc more concisely expresses control flow and communication.

Expressiveness: Integration impedance mismatch

There were a few language design impedance mismatches that we came across in integrating a data model in the Orc language. First, Orc was designed for unbounded streams of data and hence lacks a notion of detecting the end of a data stream. To handle bounded data streams such as the `ItemList` sequences produced by XQuery expressions, we designed the `GalaxListValue` wrapper class as a special kind of site that would publish a single value from the sequence when invoked with no parameters. Each invocation would step to the next element in the `ItemList` sequence. We added a fundamental site, `hasmore`, to detect if the end of the sequence had been reached. With these capabilities we developed `foreach` to apply a named expression to each value in a sequence, finally signaling when all values had been published. In Orc, all signals that aren't captured by a `where` or sequence operator instead propagate to the enclosing scope. We suggest that the Narada code could be even more succinct and readable if the Orc language added an additional type of signal corresponding to reaching the end of a bound data stream. This signal would not be captured by a sequence operator at the same scoping level, but instead would propagate to the enclosing scope. Alternatively a construct for iterating over bounded data streams would suffice if the operator could be defined with inline code instead of requiring a separate expression declaration.

Additionally, because all XML documents are encoded as XQuery sites, there is no natural way to represent globally-scoped variables like `data` that map to XML documents in Orc-X. When an XQuery site is evaluated, the published results are bound to variables whose scope is limited and must be passed explicitly to other expression definitions. In the Narada code, note that we pass all variables such as

`data` as parameters to all expression definitions that accessed or manipulated them. However this is not an ideal solution; rather we propose adding a special keyword, `var`, to the Orc language for global variable declarations. We believe that it would help improve code readability.

Finally, it is worth describing subtle coding pitfalls in complex expressions involving multi-path `if` statements. Since `if` remains silent when testing a `false` value, a section of code with a single `if` statement is not guaranteed to signal. For example, the `refresh_handler` expression in section 4.1 includes the complement of the `if(doProcess)` so the entire expression will signal through either code path. This is important because we need to ensure that the `Unlock` site is reached. In general, code containing side-effects must carefully consider the control-flow characteristics of Orc.

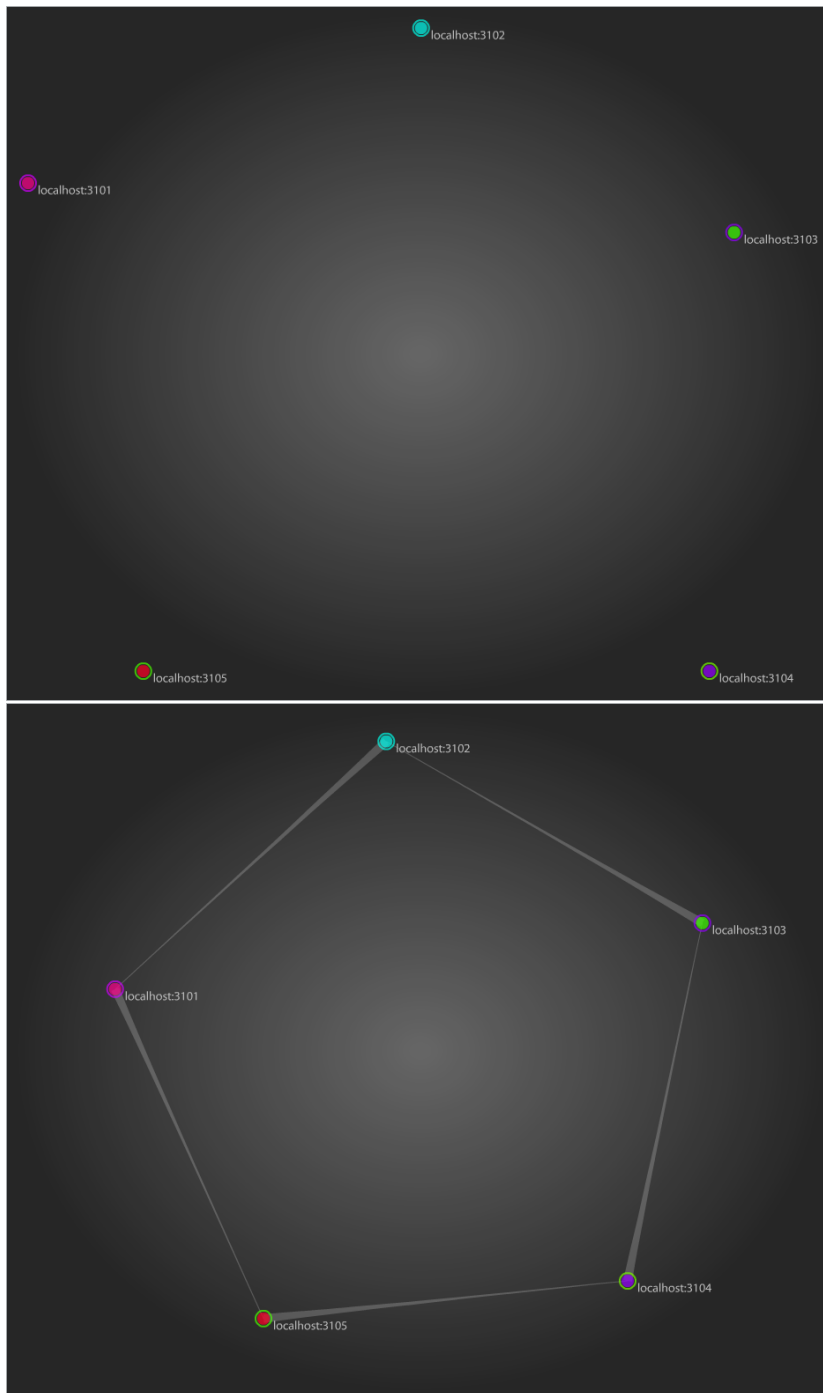


Figure 4.1: Before (top) and after (bottom) initialization

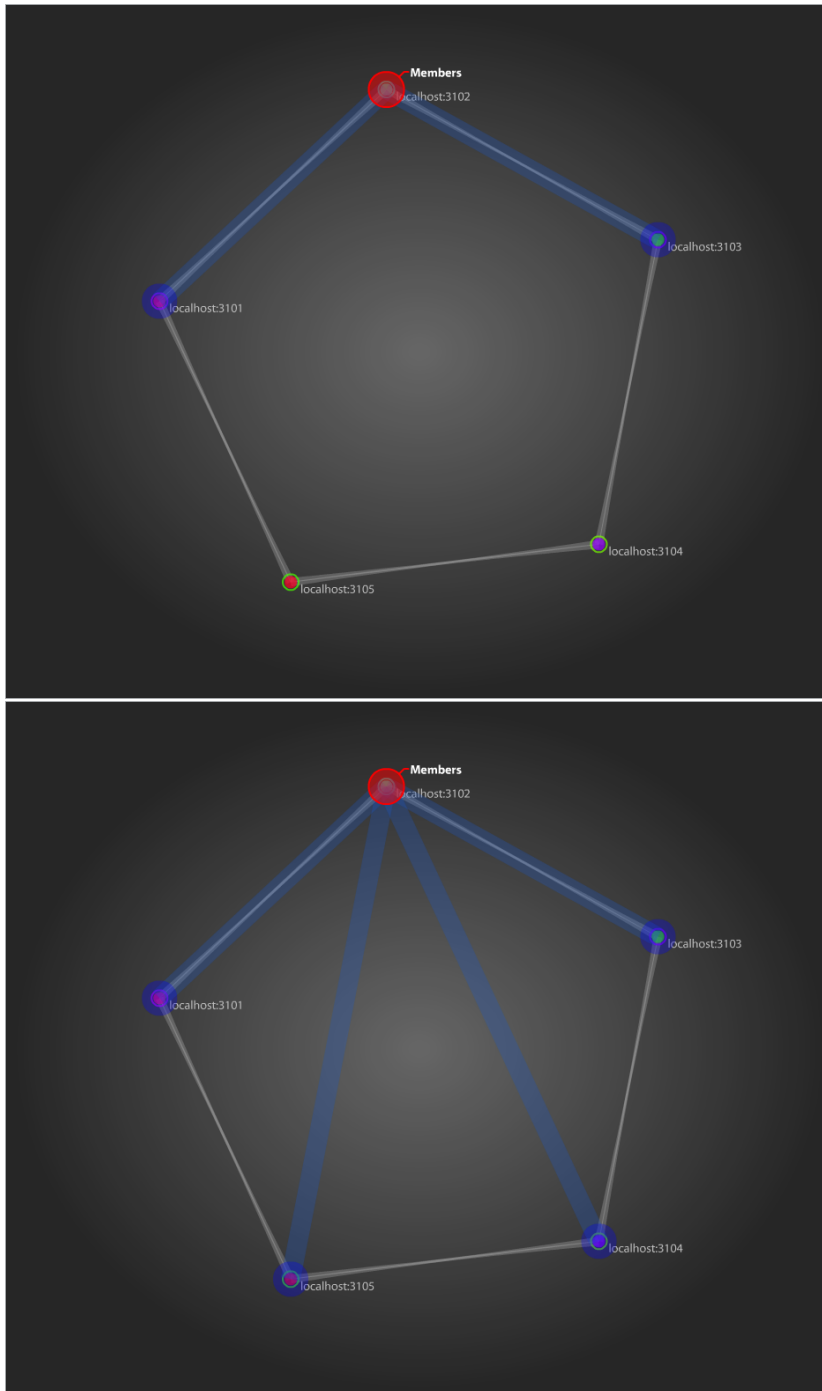


Figure 4.2: Members overlay tree: initial (top) and final (bottom)

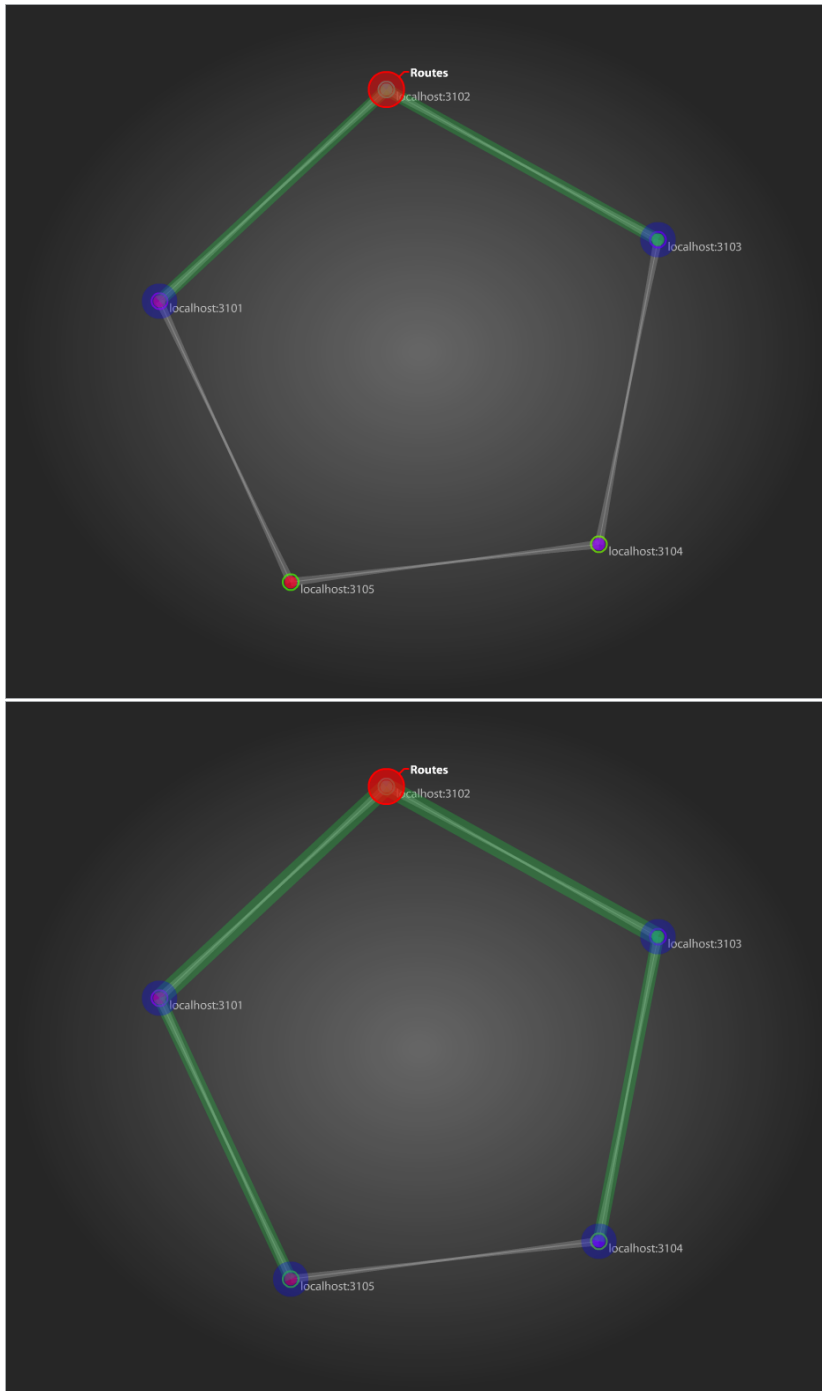


Figure 4.3: Routes overlay tree: initial (top) and final (bottom)

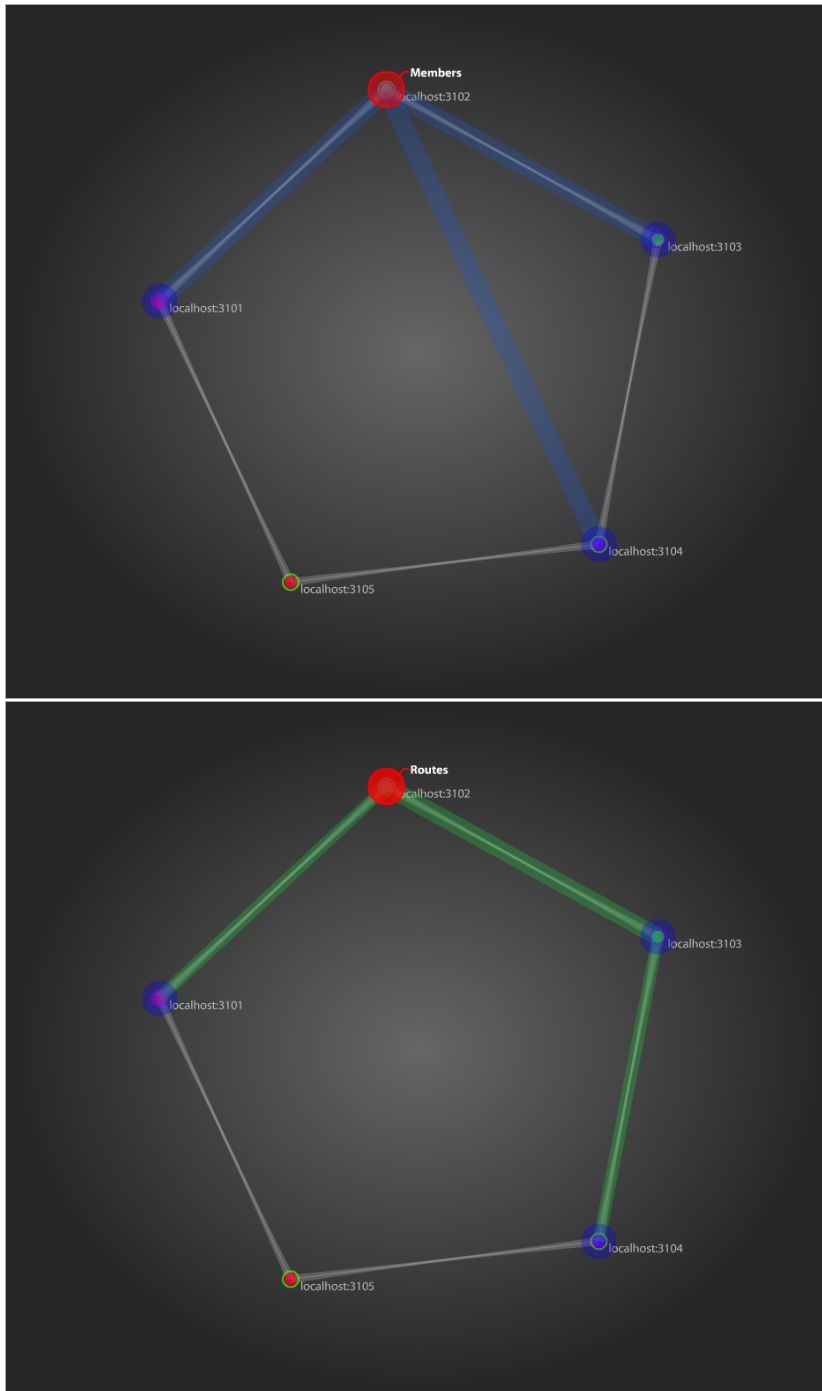


Figure 4.4: Members and routes trees after node 3105 dies

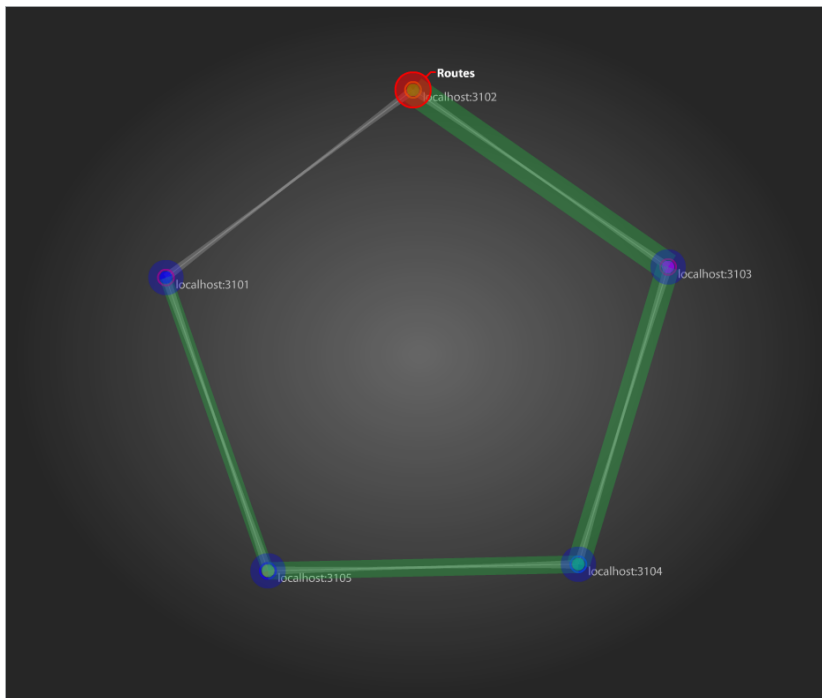


Figure 4.5: Route overlay tree with long latency between 3102 and 3101

Chapter 5

Related work

Orc-X is related to recent work on languages that have been used to implement distributed resource management protocols, and in general to languages that combine XML processing and general-purpose computation. Distributed resource management systems are traditionally implemented in imperative languages like C or Java. The languages described below have the potential to significantly reduce the code size and improve the ease of implementation of such systems. For example, the comparison of two implementations of the Domain Name Server protocol (in DXQ and C) given in [9] indicates that the relative code size is an order of magnitude larger in the C language.

There has also been work on extending XQuery to support distributed computation. The language DXQ [10, 9] extends XQuery to support distributed computing and concurrency. DXQ allows arbitrary queries to be sent to remote servers for remote execution. Unlike Orc-X, which only allows shipping XML data, DXQ allows both XML (*extensional* values) and arbitrary query plans with closures (*intensional* values) to be communicated around the network. However, DXQ has unsatisfying aspects with regard to concurrency. Unlike Orc-X, DXQ doesn't have explicit concurrency operators, which greatly limits code readability with regard to

visualizing and understanding the parallelism in the Narada algorithm. In addition, DXQ requires explicit locks for synchronization which can lead to deadlock, race conditions, and other problems typical of lock-based synchronization. This concern has not yet been fully addressed by Orc-X, however software transaction support is a work-in-progress and, coupled with the structured concurrency operators, they may help address or avoid the lock-based synchronization challenges. Furthermore, the current implementation of DXQ doesn't provide a way of expressing time-out tolerance in the case of a network failure, as Orc-X does.

In summary, Orc-X is a nice alternative to DXQ for distributed resource management protocols for several reasons. First, the structured concurrency constructs in Orc improve code readability because it makes the parallelism explicit in the algorithm. We demonstrated this in Chapter 4 in our implementation of the Narada protocol. Furthermore, Orc has the notion of time built into semantics [22]. This makes it easy to express time-outs. Having this capability is necessary for distributed resource management protocols because they operate in an environment that exhibits dynamic behavior. For example, a ping request to a remote peer may not return a result if a communication link is down.

OverLog [15] and NDlog [16] are logic programming languages that have been extended to support distributed computation. OverLog enables succinct implementation of distributed protocols such as Narada [15]. We believe that the usability of the language is open to debate – and this is not a question that can be settled by simple experiments. As in Prolog, the control flow of the system is encoded in the dependencies between clauses – whether this is an advantage or not is debatable. Our subjective evaluation is that the language does a good job of hiding the communication aspect of distributed protocols. This has the potential disadvantage of making it more difficult to specify communication patterns explicitly. Instead, it is assumed that the language compiler can automatically optimize communication.

For example, the Narada implementation in the paper [15] specifies that individual tuples are sent from one machine to another. This can lead to an n -squared number of messages, where n is the number of machines. While it might be possible to optimize this communication, this compilation capability is not described in the paper. Concurrency is also implicit in the language. As a result, OverLog abstracts away what may be considered essential aspects of the distribution problem: concurrency and communication. Very powerful compilation techniques are needed to make this work. Orc, on the other hand, provides constructs for managing concurrency in an explicit, structured way. By adding XQuery to Orc, Orc-X also gives the programmer control over the granularity of communication.

$C\omega$ [2] is an object-oriented language that includes first-class language support for three data models: object model, relational model (e.g., SQL tables) and semi-structured model (e.g., XML). The $C\omega$ research language included some significant advances around integration of XML/queries and novel concurrency models within a traditional object-oriented programming language. The query and XML capabilities have been modified and extended for inclusion in the current C# standard. The new features, called Linq, allow type-safe queries over XML [18] and relational data source [4]. It is also possible to write XML literals. The concurrency features, which are based on the Join calculus [1], have not yet been applied to the C# language. $C\omega$ provides polyphonic, asynchronous concurrency primitives [2]. Currency in $C\omega$ is based on *asynchronous messages* and *chords* [2]. Asynchronous messages introduce concurrency, because the caller can continue after invoking a concurrent service. Chords allow synchronization of multiple asynchronous calls. Like $C\omega$, Orc-X allows for asynchronous messaging (in addition to explicit synchronous messaging) and the concurrency primitives are composable. For example, asynchronous messages can be composed with the `|` and `where` constructs and synchronous messages can be composed with the `>>` construct. However, Orcs communication sites

are explicit and are already based on standard high-level communication patterns a la remote procedure calls over HTTP. We do not know of an implementation of Narada in $C\omega$. $C\omega$'s concurrency primitives are higher-level than threads and locks, but the higher-level patterns of communication in Narada would have to be encoded as patterns of asynchronous messages. Orc-X, on the other hand leverages its explicit concurrency primitives (both asynchronous and synchronous) along with explicit send and receive sites for managing communication between nodes.

For XML processing, $C\omega$ achieves the semantic behavior of XPath expressions with the *dot* operator. Similarly, Orc-X uses XPath expressions, which are part of the XQuery spec [3]. $C\omega$, unlike Orc-X, lacks XML update semantics [2], so XML documents can not be updated. Orc-X utilizes XQuery update semantics [6] to update XML documents. This capability is necessary for the Narada protocol, as it periodically makes updates to node-local state. $C\omega$ has first-class language support for the relational data model by supporting SQL. While Orc-X currently only supports the XML data model, this project is proof that Orc could be easily extended to support other data models such as SQL-like relational data. Finally, unlike Orc-X $C\omega$ lacks a notion of time in its semantics. This makes it difficult to express time-outs, which is key to facilitating the detection of failures, such as dropped connections between nodes in networked, distributed environments.

As mentioned in our discussion of the $C\omega$ language, the query and XML capabilities of $C\omega$ (also known as Linq) exist in the C# standard and hence, in the .NET framework. XLinq is an in-memory XML API extension of the Linq language that allows type-safe XML query capabilities [4] and allows for expression of XML literals [18]. DLinq is an API extension of Linq that allows for type-safe SQL query capabilities for relational data sources [4]. So, in summary, the Linq language consists of 2 primary APIs: DLinq for SQL relational data and XLinq for manipulating semi-structured XML data. Linq has no concurrency operators as a

standalone language. Since Linq was subsumed by the $C\omega$ language, Linq coexists with $C\omega$'s concurrency operators. $C\omega$'s concurrency is based on the Join calculus [1]. As we mentioned earlier, these concurrency features have not yet been applied to the $C\#$ language nor is there an implementation of the Narada protocol in this language for us to compare.

The Linq data model is of particular interest because it is an amalgam of several unique data models in one language. Like Linq, Orc-X has in-memory support for XML documents and query results. However, Linq includes an additional API for SQL-like languages, which support relational data. For the applications we chose to represent, XML is a suitable model for representing the data. Because Orc is a web services-based, communication-centric language, XML is a good model to use. We have not yet determined a need for supporting a SQL-like, relational data model. However, the Orc-X project has demonstrated the relative ease in integrating new data models with the Orc language. Hence, it is conceivable and fairly fluid to add additional data models to Orc if future applications warrant a need for such a data model. In summary, Linq's inherent weaknesses compared to Orc-X for expressing distributed resource management protocols like Narada are the same as in $C\omega$. Since Linq is a part of $C\omega$ and we do not know of an implementation of Narada in $C\omega$, we can't make any direct comparisons of our implementations.

Chapter 6

Conclusions and future work

In this thesis we have presented Orc-X, a language that combines an XML data model with Orc to express a complex, distributed resource management protocol. As we have shown in Chapter 4, Orc-X can concisely express the semantic needs of these applications through an example implementation of the Narada protocol. Orc's main virtues include efficient orchestration of computations and communication. These virtues in combination with XML/XQuery proved to be quite complimentary.

Applications like Narada operate in a dynamic environment, where the network is constantly evolving. These protocols send periodic, asynchronous messages around an ad-hoc network of nodes to manage the structure of the overlay. Because the environment is constantly evolving, applications in this domain also need to be able to detect dropped communication links. Having a notion of time-out tolerance is helpful in detecting broken or dropped communication links. The languages that are the strongest for representing the semantics needs of distributed resource management protocols must address the aforementioned concerns. Additionally, the language must have an appropriate data model. For example, each node that implements the Narada protocol must maintain local state, which includes information about neighboring nodes and the structure of the overlay network. This information

can be represented as a shallow tree. XML's in-memory representation is a tree, which is appropriate for representing the node-local state for Narada. On the other hand, if there were many relationships between nodes in the node-local state, then the SQL-like relational data model may be more appropriate for representing these cross-edge relations. Finally, since applications in this domain need explicit concurrency and communication, Orc-X is a suitable candidate for satisfying these needs. Not only does Orc-X provide an appropriate data model through XML, but it is also a communication-centric language that makes communication and concurrency explicit.

Future work

Orc-X has great potential as a language and so far has proven to be quite powerful in its ability to express distributed protocols such as the Narada protocol. Because it is not a traditional sequential language and provides language facilities for concise expression of parallel and sequential computations, it has potential to be a useful language for many other distributed resource management protocols, such as Chord [15] or Willow [21].

The development of Orc-X has touched upon some open research problems in Orc; solutions to these problems would substantially enhance the expressiveness of Orc-X.

Distribution

As currently written, the Narada protocol in Orc-X runs as separate copies of the same workflow on different nodes in the network, using explicit send/receive calls and XML messages to ship data between these nodes. However, Orc would ideally express the entire protocol in a single orchestration, rather than in many localized program instances. Distributed execution is an active area of study in Orc: how does

one write a single program and then parcel its execution out across many machines, moving code and data as necessary? If the Narada protocol were written in this way, there would be no explicit sends or receives; operations which used references to data located on different machines would either move code to that machine or automatically retrieve the data.

Synchronization and atomicity

DXQ uses explicit locks throughout the Narada implementation to ensure mutual exclusion of node-local state. As shown in research [13] explicit locking in concurrent programming is difficult to write correctly and is often subject to well-known issues such as *deadlock*, *livelock* and *priority inversion*. As stated in the implementation description, Orc-X currently assumes that XQuery expressions will execute in such a way as to avoid data corruption (the naive strategy is, of course, serialization of all embedded executions). However, this is unsatisfactory as it creates an obvious synchronization bottleneck. We are currently investigating the addition of optimistic concurrency, i.e. transactions, to the Orc semantics, in the form of a combinator `atomic f`, which runs the expression `f` atomically using a variant of software transactions. In Orc-X, we could concisely ensure atomic updates to node-local state in the Narada protocol by using `atomic` to enclose handler invocations such as `refresh_handler`.

Appendix A

Narada in Orc-X

```
-- Standard metronome definition with a time parameter
def Metronome(t) =
  Rtimer(t) >>
  (Signal | Metronome(t))

--See page 10 for explanation of forall
def forall(S) =
  S >v>
  ( let(v) | forall(S) )

--See page 11 for explanation of foreach
def foreach(S, oper, data) =
  ( (if (more) >>
    S >item>
    ((let(this, rest)
      where this in oper(item, data))
     where rest in foreach(S, oper, data)))
  |(if (~(more)) >>
    Signal))
```

```

    where more in hasmore(S)

--Helper function that increases the sequence number of local member
--(mem) by one
def nextseq(mem) =
    {replace value of node $mem/sequence with $mem/sequence + 1}

-- Advance the sequence number of this node in its own member table
def stepseq(data) =
    Lock() >>
    nextseq({$data/members/member[@address=$data/self]}) >>
    Unlock()

--Explanation on page 29
def send_message(addr, content, data) =
    Send(addr, msg)
where msg in
{<message origin="{ $data/self}">
    { $content }
</message>}

-- Helper function
-- Adds the address addr as a new neighbor to the local neighbor
--and routing tables
def addNeighbor(addr, data) =
    if(~doAdd) >>
        Signal
    | if(doAdd) >>
        (let(addneighbor, addroute)
         where addneighbor in doAddNeighbor(addr, data);
          addroute in doAddRoute(addr, data))

```

```

where doAdd in empty({$data/neighbors/neighbor[@address=$addr]})

--Helper function for addNeighbor that inserts the new neighbor into the
--local neighbor table
def doAddNeighbor(addr, data) =
  {insert node <neighbor address="{ $addr}"/> into $data/neighbors;} >>
  {$data/neighbors}

--Helper function for addNeighbor that inserts or replaces a newer route
--into the local routing table
def doAddRoute(addr, data) =
  ((if(doInsert) >>
    {insert node $new_route into $data/routes; $new_route} >>
    {$new_route})
  |(if(~doInsert) >>
    {replace node $data/routes/route[@address=$addr] with
      $new_route} >>
    {$new_route})
  where doInsert in empty(R)
  )
  where new_route in Ping(addr) >latency>
    {<route address="{ $addr}" distance="1
      latency="{ $latency}">
      <hop from="{ $addr}"/>
    </route>};
  R in {$data/routes/route[@address=$addr]}

--Explanation on page 27
def initialization(data, neighborSet) =
  {insert node <routes> <route address="{ $data/self}" distance="0"
    latency="0"/>

```

```

        </routes> into $data} >>
let(init_neighbors, init_members)
where init_members in gettime() >t>
    ({let $new_member :=
        <members>
            <member address="{ $data/self}">
            <sequence>0</sequence>
            <time>{$t}</time>
            <live>1</live>
        </member>
    </members>
    return {insert node $new_member into $data;
    $data/members};}
);
    init_neighbors in {insert node <neighbors> </neighbors>
        into $data} >>
        foreach({$neighborSet/neighbor}, addNeighbor,
            data) >>
            {$data/neighbors}

--Helper function for neighbor_probe that deletes a neighbor from the
--local neighbor table, updates the local member table to mark
--the liveness of the removed neighbor as "dead" or 0 and
--removes the neighbor from the local routing table
def removeNeighbor(addr,data) =
    Lock() >>
    (let(a,b,c) >>
        where a in {delete node $data/neighbors/neighbor[@address = $addr]} >>
            {$data/neighbors};
        b in forall({$data/members/member[@address = $addr]}) >m>
            {replace value of node $m/live with 0} >>

```

```

        nextseq(m) >>
        {$data/members};
        c in {delete node $data/routes/route[hop/@from=$addr or
        hop/@to=$addr]} >>
        {$data/routes}) >>
Unlock()

--Explanation given on page 31
def neighbor_probe(n,data) =
  forall({$data/members/member[$n/@address = @address]}) >m>
    if(doRemove) >>
      removeNeighbor(addr,data)
  | if(~doRemove) >>
    Signal
  where doRemove in boolValue({$dt > $thold});
        dt in gettime() >current> {$current - $m/time};
        addr in {$n/@address};
        thold in {30}

--Helper function that selects a random member from the local member
--table
def random(S) =
  {let $memct := fn:count($S) - 1
  return
    if ($memct > 0) then
      let $r := glx:random_int($memct) + 2,
          $y := $S[$r]
      return $y
    else ();} >>
  {$y}

```

```

--Helper function for latency_probe that replaces or inserts
-- the new latency into the local member table
def setLatency(m, latency) =
  Lock() >>
  (((if (~doInsert) >>
    {replace value of node $m/latency with $latency})
  |(if (doInsert) >>
    {insert node <latency>{$latency}</latency> into $m})) >>
  where doInsert in empty({$m/latency})) >>
  Unlock()

def removeMember(addr,data) =
  removeNeighbor(addr,data)

--Explanation given on page 30
def latency_probe(m,data) =
  if(boolValue({$latency < $limit})) >>
    setLatency(m, latency)
| if(boolValue({$latency >= $limit})) >>
  removeMember(m)
where latency in (Ping(addr) | Rtimer(orLimit) ) >> {$limit};
  limit in {100};
  addr in {$m/@address};
  orLimit in let(1000)

--Explanation given on page 30
def refresh_probe(n,data) =
  send_message(n, content, data)
  where content in
    {<refresh>
      {$data/members}

```

```

    </refresh>}

--Explanation given on page 29
def routing_probe(n,data) =
  send_message(n, content, data) >>
  where content in
    {<routing>
      {$data/routes}
    </routing>}

--Helper for refresh_handler that inserts a new member into the
--local member table
def insertMember(r_m,data) =
  {insert node
    <member address="{r_m/@address}">
      <sequence>{r_m/sequence}</sequence>
      <time>{t}</time>
      <live>{r_m/live}</live>
    </member>
  into $data/members}
  where t in gettime()

--Helper for refresh_handler that replaces the sequence, time and
--live fields in the local member table with more recent values
def updateMember(l_m, r_m) =
  (if(~doUpdate) >>
    Signal
  |(if(doUpdate) >>
    {replace value of node $l_m/sequence with $r_m/sequence} >>
    {replace value of node $l_m/time with $t} >>
    {replace value of node $l_m/live with $r_m/live}
  )

```

```

    where t in gettime()))
where doUpdate in
    boolValue({$l_m/sequence < $r_m/sequence})

--Explanation given on page 33
def refresh_handler(neighbor_members,data) =
    forall({$neighbor_members/member}) >r_m>
        {$r_m/@address} >addr>
        Lock() >>
        ( if(~doProcess) >>
            Signal
        | if(doProcess) >>
            ( if(doInsert) >>
                insertMember(r_m,data)
            | if(~doInsert) >>
                updateMember(l_m, r_m)
            where doInsert in empty(l_m);
                l_m in {$data/members/member[@address=$addr]}
            )
        where doProcess in boolValue({$data/self != $addr})
        ) >>
        Unlock()

--Helper function for routing_handler that determines the latency
--to a given neighbor
def setup(neighbor, data) =
    {$data/routes/route[@address=$neighbor]} >route_to_neighbor>
    (( if(empty(route_to_neighbor)) >> Ping(neighbor))
    |( if(~empty(route_to_neighbor)) >> {$route_to_neighbor/@latency} )
    )

```

--Explanation given on page 33

```
def routing_handler(neighbor,neighbor_routes,data) =
  setup(neighbor, data) >latency>
  forall({$neighbor_routes/route}) >nr>
    if(~empty(nr)) >>
      if(empty({$nr/hop[(@to|@from)=$data/self]})) >>
        Lock() >>
          ( {$data/routes/route[@address = $nr/@address]} >Routes>
            ( if(doInsert) >>
              {insert node $new_route into $data/routes}
            | if(~doInsert) >>
              ( if(doReplace) >>
                {replace node $Routes with $new_route}
              | if(~doReplace) >>
                Signal
              )
            where doReplace in
              boolValue({$nr/@distance + 1 < $Routes/@distance})
          )
        where doInsert in empty(Routes);
          new_route in
            {$nr/hop[1]} >fst_hop>

            {attribute to {$fst_hop/@from}} >to_attr>
            {<route address="{ $nr/@address}"
              distance="{ $nr/@distance + 1}"
              latency="{ $latency + $nr/@latency}">
              {(<hop from="{ $neighbor}">{$to_attr}</hop>, $nr/hop)}
            </route>}
          ) >> Unlock()
```

--Explanation given on page 28

```
def Neighbors(data) =  
  forall({$data/neighbors/neighbor})
```

--Explanation given on page 28

```
def neighbor_metronome(data) =  
  Metronome(9000) >>  
  Neighbors(data) >n>  
  neighbor_probe(n,data)
```

--Explanation given on page 28

```
def latency_metronome(data) =  
  Metronome(7000) >>  
  random({$data/members/member}) >m>  
  if(~empty(m)) >>  
  latency_probe(m,data)
```

--Explanation given on page 28

```
def refresh_metronome(data) =  
  Metronome(3000) >>  
  stepseq(data) >>  
  Neighbors(data) >n>  
  refresh_probe({$n/@address},data)
```

--Explanation given on page 28

```
def routing_metronome(data) =  
  Metronome(5000) >>  
  Neighbors(data) >n>  
  routing_probe({$n/@address},data)
```

--Explanation given on page 32

```

def listen_for_messages(data) =
  forall(Receive) >msg>
    {$msg/@origin} >origin>
    if(~empty(origin)) >>
      ( {$msg/refresh/members} >m>
        if(~empty(m)) >>
          refresh_handler(m,data)
        | {$msg/routing/routes} >r>
          if(~empty(r)) >>
            routing_handler(origin,r,data)
        | Lock() >>
          addNeighbor(origin,data) >>
          Unlock()
      )

-- The Narada Protocol --
--Explanation given on page 25
{<data>
<self>localhost:3101</self>
<name>n1</name>
<colors green="2150464" blue="2119873" yellow="15464765" />
</data>} >data>
{<neighbors><neighbor>localhost:3104</neighbor>
      <neighbor>localhost:3102</neighbor>
    </neighbors>} >neighborSet>
initialization(data, neighborSet) >>
( listen_for_messages(data)
  | refresh_metronome(data)
  | latency_metronome(data)
  | neighbor_metronome(data)
  | routing_metronome(data))

```

Bibliography

- [1] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C. In *Proceedings of ECOOP02*, pages 415–440, 2002.
- [2] G. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in Comega. In *Proceedings of ECOOP*, 2005.
- [3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, 2007. <http://www.w3.org/TR/xquery>.
- [4] D. Box and A. Hejlsberg. Linq Project Overview, 2007. <http://msdn.microsoft.com/netframework/future/linq>.
- [5] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium World Wide Web Consortium, September 2006. <http://www.w3.org/TR/xml/>.
- [6] D. Chamberlin, D. Florescu, J. Melton, J. Robie, and J. Simeon. XQuery Update Facility. Technical report, World Wide Web Consortium, 2007. <http://www.w3.org/TR/xquery-update-10/>.
- [7] Y.-H. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), 2002.

- [8] O. Dahl, E. Dijkstra, and T. Hoare. *Structured Programming*. ACM Classic Books Series, 1972.
- [9] M. Fernández, T. Jim, K. Morton, N. Onose, and J. Siméon. DXQ: A Distributed XQuery Scripting Language. In *Proceedings of the SIGMOD Workshop on XQuery Implementation and Practice (XIME-P)*, 2007.
- [10] M. Fernández, T. Jim, K. Morton, N. Onose, and J. Siméon. Highly Distributed XQuery with DXQ. In *Proceedings of ACM Conference on Management of Data (SIGMOD), Demonstration Program.*, June 2007.
- [11] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C recommendation. Technical report, 2007. <http://www.w3.org/TR/xpath-datamodel/>.
- [12] Galax: The XQuery Implementation for Discriminating Hackers.
- [13] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
- [14] T. Hoare. Structured Concurrent Programming. October 2007.
- [15] B. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SIGOPS Oper. Syst. Rev*, 2005.
- [16] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, pages 97–108, New York, NY, USA, 2006. ACM Press.
- [17] W. Meier. exist: An open source native xml database. In *Revised Papers From*

the Node 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems, 2002.

- [18] E. Meijer and B. Beckman. XLinq: XML Programming Refactored (the return of the monoids). In *Proceedings of XML*, 2005.
- [19] J. Misra and W. R. Cook. Computation Orchestration: A Basis for Wide-Area Computing. In *Journal of Software and Systems Modeling*, 2006.
- [20] K. O’Hair. HPROF: A Heap/CPU Profiling Tool in J2SE 5.0. <http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>, November 2004.
- [21] R. van Renesse and A. Bozdog. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS ’04)*, Feb. 2004.
- [22] I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. A Timed Semantics of Orc. Submitted for publication to *Theoretical Computer Science*, 2007.

Vita

Kristi Morton was born in Garland, Texas on June 22, 1981 to Buddy and Marissa DiBennardo. She graduated from Garland High School in Garland, Texas in 1999 and matriculated shortly thereafter at Rice University, Houston, Texas. In January 2003 she received the Bachelor of Art degree in Computer Science. From 2003 to 2006 she worked at Freescale Semiconductor (formerly Motorola) on the GNU C Compiler. The summer prior to entering graduate school, Ms. Morton interned at AT&T Labs in Florham Park, NJ, and contributed to the Distributed XQuery research effort.

Permanent Address: 11913 Misty Brook Dr.

Austin, TX 78727

kristid@alumni.rice.edu

This thesis was typeset with $\text{\LaTeX} 2_{\epsilon}$ ¹ by the author.

¹ $\text{\LaTeX} 2_{\epsilon}$ is an extension of \LaTeX . \LaTeX is a collection of macros for $\text{T}_{\text{E}}\text{X}$. $\text{T}_{\text{E}}\text{X}$ is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.