# Orc-X: Combining Orchestrations and XQuery

## (work in progress)

Kristi Morton David Kitchin William Cook

University of Texas at Austin

{kmorton, dkitchin, wcook}@cs.utexas.edu

## Abstract

In designing a language for distributed computing, the handling of *data* and *distribution* can be viewed as largely orthogonal concerns, as long as the data model supports the communication requirements of the distribution model. This view contrasts strongly with approaches based on distributed objects, which typically enforce a tight coupling of state and behavior. We have previously presented Orc, a language that provides simple but powerful constructs to orchestrate distributed computations. Previous versions of Orc included only simple data types, since these were sufficient to demonstrate the concurrency primitives. However, Orc's 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. 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 the application domain of distributed resource management protocols such as the Narada mesh overlay protocol.

***Categories and Subject Descriptors*** D.1.3 [*Concurrent Programming*]: Distributed programming

***General Terms*** Languages, Design

***Keywords*** XQuery, Orc, orchestration, Narada

## 1. Introduction

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. 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. They should also support evolution, by allowing a single node to be upgraded without bringing down the complete network. Current programming languages and models do not facilitate building these kinds of distributed data-oriented applications.

The Orc language is a domain-specific language that targets applications with distributed and concurrent computations. Orc's combinators aggregate invocations of services into orchestrated, structured concurrent programs. *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.

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.

We present Orc-X, an extension of the Orc programming language which enhances structured concurrency compositions from Orc [18] with XML data and XQuery processing. XQuery [3] is a declarative query language for XML-based applications. XQuery leverages XML for representing semi-structured data as well as efficiently querying XML data.

We demonstrate Orc-X by showing an example implementation of the Narada overlay protocol. This distributed protocol entails sending periodic, asynchronous messages around an overlay network for managing its structure. The structure of the overlay network is encoded in the node-local state, which includes neighbor and group memberships as well as routing tables. This information is naturally represented in XML and we leverage XQuery to query and manage the semi-structured XML data at each node. Furthermore, since the Narada 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, the Narada implementation demonstrates the potential of Orc's combinators, which concisely and elegantly express distributed computations.

**Organization** The rest of this proposal is organized as follows. Section 2 presents an introduction to Orc, XQuery and Narada. Section 3 presents an introduction to the language concepts of Orc-X and section 4 discusses our Narada implementation. Section 5 presents related work and section 6 discusses the challenges in combining Orc with XQuery as well as the interesting future challenges of Orc-X in other application domains.

*Technical Report TR-07-63*

## 2. Background

### 2.1 Orc

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.

### 2.2 Examples

#### 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 how time-outs are used to mitigate this problem. Because Orc has time built into its semantics [20], 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 `more?(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) >> Signal
            where this in oper(item)
            where rest in foreach(S, oper) )
  | if (!more) >>
      Signal
 where more in more?(S)
```

## 2.3 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 section 4, iterators are constructed with sequence sites.

## 2.4 Narada

Narada is a distributed, asynchronous, self-organizing protocol for managing the structure of an overlay network. This infrastructure is used for routing and end-system multicast purposes [7]. The overlay network contains 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 ensure current latencies and to determine 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.

## 3. Orc-X Overview

The Orc-X language combines Orc with XML and XQuery. Orc is the *master language*, which manages all control flow and data flow. XQuery expressions are embedded inside of Orc expressions to manage the data in our XML data model using queries and update operations. In this section, we discuss the syntax and semantics of Orc-X as an extension of Orc. Orc-X is a prototype for a general approach to embedding both application-specific and general-purpose languages within Orc.

### 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 introduces a new base expression {`e`} into Orc, where `e` is an XQuery expression. `e` is *open*; it may contain free variables that are not defined within the expression itself. Those free variables correspond to Orc variables in the scope surrounding the expression.

An embedded XQuery expression may run when all of its free variables have been bound. 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]. We use the techniques shown in Section 2.2 to manipulate sequences in Orc.

### 3.2 Encoding of Orc-X into 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`($\overline{x}$). 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 sake of consistency, the variable names $\overline{x}$ used in Orc are the same names that appear in the XQuery code.

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 that 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.

## 4. Implementing Narada with Orc-X

In this section, we summarize our implementation of the Narada protocol in Orc-X. This is by no means an exhaustive study of Orc-X's complete capabilities; it should be viewed as a work in progress.

### 4.1 Network primitives

Currently, in order to build distributed Orc applications, Orc expressions execute locally at each network node and communicate with each other via a network model based on site calls. 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,data)` serializes the data item `data` 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.

## 4.2 Implementing the Narada protocol

We implement the Narada protocol as an Orc expression definition

```
def Narada(data, self, neighborSet) = ...
```

`data` is a reference to a fresh XML document for this node's state. `self` is the machine address for this node. `neighborSet` is the sequence of addresses for the immediate neighbors of this node.

### Core protocol

The core expression defining the entire protocol is simply

```
initialization >>
 ( listen_for_messages
 | refresh_metronome
 | latency_metronome
 | neighbor_metronome
 | routing_metronome)
```

The `initialization` expression prepares the local `data` 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, and also begins to send four different types of protocol messages at regular intervals.

### Initialization

We initiate the algorithm 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 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 =
  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}
  where init-neighbors in
        {insert <neighbors> </neighbors>
         into $data} >>
         foreach(neighborSet, addNeighbor)
  where init-members in
        {insert <local_members>
                  <member address={$self}>
                    <sequence> "0" </sequence>
                    <live>"1"</live>
                  </member>
                </local_members>
         into $data}
```

### Metronomes

The core of the Narada protocol is a small set of periodic, asynchronous probes of immediate neighbors. There are four such probe actions:

1. A routing probe sends updated routing information 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 =
  forall({$data/neighbors/neighbor/@address})

def routing_metronome =
  Metronome(5) >>
    Neighbors >n> routing_probe(n)

def refresh_metronome =
  Metronome(3) >>
    Neighbors >n> refresh_probe(n)

def latency_metronome =
  Metronome(2) >>
    random({$data/local_members/member}) >m>
      latency_probe(m)

def neighbor_metronome =
  Metronome(5) >>
    Neighbors >n> neighbor_probe(n)
```

We show implementation details for all four probes.

### Routing probe: broadcasting local routing data

We implement `routing_probe` by using the `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) =
  send(addr, msg)
    where msg in
      {return <message origin="{$self}">
                 { $content }
              </message>}

def routing_probe(n) =
  send_message(n, content)
    where content in
      {return <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.

```
def refresh_probe =
  send_message(n, content)
    where content in
      {return <refresh>
                  {$data/local_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) =
  if(latency < limit) >>
    setLatency(m, latency)
| if(latency >= limit) >>
    removeMember(m)
 where latency in ( ping(addr) | Rtimer(limit) )
 where limit in let(1000)
 where addr in {$m/@address}
```

**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 threshhold then the neighbor is removed from the node's local neighbor table and marked inactive in the local member table.

```
def neighbor_probe(n) =
  forall({$data/local_members/member[@address=$addr]}) >m>
    if(dt > thold) >> removeNeighbor(addr)
      where dt in gettime() - t
      where t in {$m/time}
      where addr in {$n/@address}
      where thold in let(60)
```

**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 if it were a sequence, 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 =
  forall(receive) >msg>
  ( {$msg/message/@origin} >orig>
      ( {$msg/message/refresh} >r>
          refresh_handler(orig, r)
      | {$msg/message/routing} >r>
          routing_handler(orig, r)
      | addNeighbor(orig) ) )
```

There is a synchronization issue here: since we continue listening for further messages to handle in parallel with the receipt of any message, it is possible for multiple message handlers to be updating the local state in parallel. We do not write explicit locking in the Orc or XQuery code; instead we assume that the implementation will serialize embedded XQuery executions so as to prevent corruption of data. This is a strong assumption; we discuss the reasoning behind this assumption, and proposals to weaken it, in Section 6.

## 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.

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.

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 serves 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.

The C$\omega$ [2] 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 [17] 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. Currency in C$\omega$ is based on *asynchronous messages* and *chords*. Asynchronous messages introduce concurrency, because the caller can continue after invoking a concurrent service. Chords allow synchronization of multiple asynchronous calls. 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.

## 6. Evaluation

Since the full implementation of Orc-X is still a work in progress, we propose methods to ensure the progress of our language. Our implementation of Orc-X will be tested by simulating the Narada protocol. We do so by examining the behavior of our implementation of the Narada protocol relative to DXQ's implementation. We know our language is on the right track if it yields the same network configuration as the DXQ implementation.

We chose to compare ourselves against DXQ for several reasons. First, DXQ has a publicly-available, full implementation of Narada. Also, Orc-X and DXQ leverage the same underlying query engine infrastructure (Galax [12]). This allows us to not only test the correctness of our algorithm but also gives us a chance to do a performance comparison.

### 6.1 Open research problems

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] explict 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`.

### 6.2 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 [19].

## A. Appendix: Narada Implementation in Orc-X

All of the example Narada code described in Section 4.2 and the full implementation of the protocol is accessible at:

`http://www.cs.utexas.edu/~kmorton/orc-x.html`

## Acknowledgments

We would like to thank the following people for their helpful comments on our paper: Mary Fernandez, Trevor Jim, Jayadev Misra and Robert Morton.

## References

[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 *In Proc. 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. Fernàndez, T. Jim, K. Morton, N. Onose, and J. Siméon. Dxq: A distributed xquery scripting language. In *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 theTwentiethAnnual 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] E. Meijer and B. Beckman. Xlinq: Xml programming refactored (the return of the monoids). In *Proceedings of XML*, 2005.

[18] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. In *Journal of Software and Systems Modeling*, 2006.

[19] 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.

[20] I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. A timed semantics of orc. Submitted for publication to Theoretical Computer Science, 2007.