Thesis Proposal

Locally Distributed Predicates: A Programming Facility for Distributed State Detection

Michael De Rosa
Carnegie Mellon University
mderosa@cs.cmu.edu

Thesis Committee: Peter Lee (Co-chair), Seth Goldstein (Co-chair), Garth Gibson, and Maurice Herlihy (Brown University)

1 Introduction

The rise of high-performance computing and internet-scale applications has spurred a renewed interest in distributed computing. Such distributed applications can range from multi-hop routing algorithms used in wireless mesh networks [22] to volunteer-driven parallel data analysis efforts [3,26]. The property of distributed systems that makes them both powerful and challenging is that they are composed of multiple autonomous, interconnected entities. Managing the interactions between these entities is the primary challenge of distributed application development.

Distributed programming is inherently more difficult than single-threaded applications, due to two complicating factors. The first of these is the need for an executing thread of a distributed program located at one computational node to access state located at a different node. Accessing remote state information is quantitatively different than accessing local memory, in that latency and failure rates are much higher in the remote case. What makes accessing remote state qualitatively different is that remote state can be modified by another component of the distributed system, raising issues of consistent data state. This is further complicated by the second factor, asynchronicity. As multiple distributed components may operate independently, there is no guarantee that state obtained from multiple nodes represents a consistent view of the distributed process, as varying rates of computation and message latency can result in the reception of out-of-date information.

To overcome these difficulties, a set of general techniques for inspecting the state of a distributed system have been developed. The most widely used technique is that of a distributed snapshot [8], in which the state of the entire system is captured in a *consistent snapshot*, which reflects a possible serialization of the parallel event stream of the distributed nodes. Distributed snapshots are a useful tool, but they require a centralized aggregation point, and are typically quite heavy-weight. In many cases, the complete state of a distributed system is unnecessary; rather it is some property of the state that one wishes to test. Development of techniques for evaluating these tests produced global predicate evaluation [12], which allows a program to evaluate a single predicate over the entire distributed system.

While global predicates allow a programmer to encode queries over the state of an entire distributed system, in distributed systems with a sparse, multihop communications topology there exists another class of distributed predicates, which we call *locally distributed predicates*. These are predicates over the local neighborhood of a particular node, bounded to a finite number of communication hops. These locally distributed predicates allow a programmer to describe the state configuration of a bounded subgraph of a distributed system. Locally distributed predicates differ from global predicates in two important respects. An important difference is that, as locally distributed predicates do not encompass the entire distributed system, there may be multiple matching subgraphs for a particular predicate. Another difference is that a locally distributed property can describe not only the logical state of the entities in a distributed system, but also their topological configuration, a property that is inherently ignored by global predicates. This prevents global predicates from efficiently detecting predicates that rely on the state and topology of a small number of nodes. Such locally distributed predicates are important in a variety of applications, including distributed debugging, multi-entity coordination, and resource discovery.

I have developed an initial set of algorithms to represent and detect locally distributed predicates in the context of debugging large ensembles of modular robots [15]. These predicates allow programmers to describe the distributed state subsets that signify an error. I have developed a simple representation language that can express logical, topological, and temporal relationships between multiple communicating robots.

My thesis is: Locally distributed predicate search provides an efficient and expressive means of evaluating locally distributed properties, and that this class of properties is useful across distributed application domains.

To validate my thesis, I must show two premises: that locally distributed predicates (LDPs) are useful in a variety of application domains, and that techniques for describing and detecting these LDPs are both theoretically sound and empirically efficient. To demonstrate the utility of LDPs, I will extend the current distributed watchpoint language with new constructs to increase the expressiveness of the system, then show that this improved system is applicable to a broad range of application domains. To demonstrate the viability of the technique, I will divide my efforts into theoretical and empirical validation. To validate the theoretical properties of LDPs, I will develop a formal grammar and semantics for the system, and use this representation to derive proof properties for LDPs in various classes of distributed systems. To experimentally validate LDPs, I plan to evaluate implementations of several distributed applications in both LDPs and existing low-level languages. I also anticipate developing one or more optimizations to improve the performance of the LDP system.

The rest of the thesis proposal is organized as follows. Section 2 describes related work. Section 3 summarizes my preliminary work on distributed watchpoint algorithms. Section 4 discusses my proposed future work, which will generalize the algorithms developed for distributed watchpoints and place them on a more sound footing. Finally, Section 5 shows my proposed schedule for completing the thesis.

2 Related Work

Research on programming distributed systems has a long and varied history, ranging from early theoretical results to more recent implementations on systems ranging from tens to thousands of nodes. Below, I summarize some of the more important and/or relevant works in relation to distributed predicate search.

2.1 Distributed Snapshot Algorithms

Lamport, in his landmark 1978 paper [24], described the inherent problems of event ordering in asynchronous distributed systems. The "happens-before" relationship described in this paper provides a weak ordering on event serialization, based on message ordering and logical clocks. Later work on vector clocks [28] provided a more optimal (if expensive) approximation of global time. These two techniques gave rise to the technique of distributed snapshots, first described in [8], which allowed for centralized capture of a *possible* global state.

2.2 Predicate Detection

In many cases, aggregating state information from every process in a distributed system represents an unnecessary expense. In particular, the detection of global boolean properties of a distributed system can be performed using global predicate evaluation (GPE) algorithms. A key notion in the study of GPE is the notion of predicates being *possibly* true (for some serialization of events) versus *definitely* true (for all serializations of events) [35]. Early GPE algorithms [12] relied on the construction of state lattices, which required exponential time to evaluate, and in fact the problem of evaluating a general global predicate was shown to be NP-complete [10]. Efficient algorithms for detecting the observer-independent [9] and conjunctive local [17, 20] subclasses of predicates have been developed.

Recently, the use of global predicate evaluation has been proposed for the detection of traffic anomalies on the Internet (as might result from a distributed denial of service attack) [21]. This has led to the creation of a triggering method based on distributed principal components analysis [19]. The detection of global predicates provides important inspiration and techniques for the detection of locally distributed predicates.

2.3 Distributed Debugging

One prominent use for distributed snapshot and predicate evaluation algorithms has been in the field of distributed debugging. Special-purpose algorithms for detecting such global properties as deadlock [32] and termination [36] have been developed. The problem of automatically detecting race conditions has also received attention, but with less success (as it is NP-complete) [7]. More general purpose distributed debugging tools, which provide message tracing and visualization, are now available as commercial tools [1]. These tools are all highly specialized, and targeted towards systems that have at most tens or hundreds of nodes.

2.4 Distributed Programming Techniques

There have been many diverse attempts at creating programming languages for distributed systems. The linear temporal logic [34] has been used to model such systems [25], and even as the foundation for a multirobot programming environment [23]. Another logic-based approach is the use of Prolog-style declarative semantics for programming distributed systems (especially overlay network algorithms) [4,27]. Process calculi such as CSP [18], the pi-calculus [29], and the ambient calculus [33] have also been used to model and program distributed systems. The abstraction of a distributed process as a set of operations on multiple data streams has been used to program both large-scale database applications [11] and wireless sensor networks [31].

3 Preliminary Work

3.1 Distributed Debugging

Designing algorithms for distributed systems is a difficult and error-prone process. Concurrency, non-deterministic timing, and state-space explosions all contribute to the likelihood of bugs in even the most meticulously designed software. These factors also make the detection of such bugs very difficult.

Tools to assist programmers in debugging distributed algorithms are few and generally inadequate. Most are forced to fall back on standard debugging methods, such as single-node debuggers (e.g. GDB [16]) or logging through console I/O such as printf(). Both debuggers and logging must be used very cautiously, or their file/console I/O can impose unintended serialization (altering the timing behavior of the ensemble) and possibly mask some bug manifestations. Debuggers are useful for detecting errors local to an individual thread or process but are not effective for errors resulting from the interactions or states of multiple threads of execution that span multiple nodes. Console or file logging may be used to detect some of these errors, but this requires collecting all potentially relevant state information at each node. The information must then be centrally collected, correlated, and post-processed, in order to extract the details of the error condition. This requires significant effort, skill, and even luck on the part of the programmer.

One would like to have a tool that can allow programmers to specify and detect *distributed conditions* in an ensemble. Such conditions constitute logical relationships between state variables that are distributed both temporally and spatially across the ensemble. Generally, distributed conditions cannot be detected by observing the state of any single node, or even the whole system at any single time. A tool which can detect such conditions can provide insights into the logical and temporal behavior of the systems and help pinpoint defects in distributed algorithms.

Unfortunately, detecting arbitrary distributed conditions which may range freely inside an ensemble of nodes can be quite difficult, given the large number of node subsets that must be examined. As an example, a condition which occurs in a subset of three nodes within an ensemble of one thousand nodes would require examining almost one billion possible size-3 subsets. To avoid this problem, we limit ourselves to the subclass of conditions which can be detected within *fixed-size, connected subensembles* of the entire ensemble. That is to say, there must be at least one communications link between each element of the subensemble. Taking the above example, given a condition involving three nodes, in a topology where each node has at most four neighbors, one would have to examine at most 4 million different subensembles, a reduction of almost 3 orders of magnitude.

To this end, we have developed a *distributed watchpoint* mechanism, which easily permits programmers to specify spatio-temporal conditions across an ensemble of nodes and trigger debugging actions when they are detected. We have developed a fully-distributed, online detection algorithm that is applicable to distributed condition detection in both simulated distributed systems, as well as physical hardware.

3.2 Describing Errors

The first step in detecting distributed error conditions is being able to represent them concisely. To that end, we have created a simple watchpoint description language, based on a fragment of linear temporal logic with the addition of predicates for state variable comparison and topological restriction. Linear temporal logic serves as the inspiration for this simple descriptive syntax and provides meaningful semantics for several of the operators.

3.2.1 Abstract Model

We consider a simplified model of a generalized distributed system, composed of an ensemble of computational nodes, connected by bidirectional communications links. Each node has some set of named state variables, and a unique id number. Nodes can communicate with other nodes only via their communications channels, which are FIFO and reliable but otherwise unconstrained. We assume that each node iterates through three atomic phases: computation, state-variable assignment, and communication. Each phase may take an arbitrary amount of time. Furthermore, we assume that each node has a copy of the watchpoint, and that each node has the state variables needed by the watchpoint expression. We explicitly do not require that all nodes have the same set of state variables, merely that they contain

$$
\begin{array}{rcl}
\langle\text{watchpoint}\rangle & \longrightarrow & \langle\text{module decl.}\rangle\ \langle\text{bool}\rangle \\
\langle\text{module decl.}\rangle & \longrightarrow & \texttt{modules}\ \texttt{(}\ \langle\textit{string}\rangle^{+}\ \texttt{)} \\
\langle\text{bool}\rangle & \longrightarrow & \texttt{not}\ \langle\text{bool}\rangle \\
 & & \mid\ \langle\text{bool}\rangle\ \texttt{and}\ \langle\text{bool}\rangle \\
 & & \mid\ \langle\text{bool}\rangle\ \texttt{or}\ \langle\text{bool}\rangle \\
 & & \mid\ \texttt{neighbor}\ \texttt{(}\ \langle\text{module}\rangle\ \langle\text{module}\rangle\ \texttt{)} \\
 & & \mid\ \texttt{(}\ \langle\text{compare}\rangle\ \texttt{)} \\
 & & \mid\ \texttt{(}\ \langle\text{bool}\rangle\ \texttt{)} \\
\langle\text{module}\rangle & \longrightarrow & \langle\textit{string}\rangle \\
\langle\text{compare}\rangle & \longrightarrow & \langle\text{numeric}\rangle\ \langle\text{c-op}\rangle\ \langle\text{numeric}\rangle \\
\langle\text{numeric}\rangle & \longrightarrow & \langle\text{state var}\rangle \\
 & & \mid\ \langle\textit{integer}\rangle \\
 & & \mid\ (\langle\text{numeric}\rangle\ \langle\text{m-op}\rangle\ \langle\text{numeric}\rangle) \\
\langle\text{state var}\rangle & \longrightarrow & \langle\text{module}\rangle\ \texttt{.}\ (\texttt{last.} \mid \texttt{next.})^{+}\ \langle\textit{string}\rangle \\
\langle\text{c-op}\rangle & \longrightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{<=} \\
\langle\text{m-op}\rangle & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \star \mid \texttt{/}
\end{array}
$$

Figure 1. Extended BNF grammar for watchpoint description language

all variables required by the watchpoint. This simplified model does not entail loss of generality, as we can express run-loop, finite-automata, or event-driven distributed programs using it.

3.2.2 Watchpoint Expressions

To reiterate what was mentioned in Section 3.1, in representing distributed error conditions, we make a key assumption: *the error must be able to be represented by a fixed-size, connected subensemble of nodes in specific states.* This assumption makes the detection of distributed error conditions feasible. Allowing disconnected subensembles would imply an exponential search through all subsets of the total ensemble, and distributing information between the members of these subsets would require significant multi-hop messaging.

Watchpoint descriptions begin with a named list of node variables. This list defines the size of the matching subensemble (the connected subgroup whose state satisfies the watchpoint expression), and is implicitly quantified over all connected subgroups of this size in the ensemble. The language includes the standard boolean and grouping primitives, basic mathematical operators, topological restrictions, and state variable comparisons. Topological restrictions take the form of the predicate `neighbor(a b)`, and indicate that the two specified nodes are neighbors (that they share a communications link). When multiple neighbor expressions are combined using boolean operators, arbitrary node topologies can be expressed. State variable comparisons allow for the comparison of named state variables in one node against constants, other local variables, or remote variables on other nodes. Additionally, state variable comparisons may include arbitrarily nested uses of the `last` and `next` temporal modal operators, which provide access to the past and future states of the node's state variables. In the case of the `next` operator, this implies that the watchpoint triggers in the "future," and that the state of the nodes would need to be rolled back one or more timesteps when the watchpoint triggers. A full formal syntax of the watchpoint description language, and several examples of its use, are provided in [15].

These simple primitives give us the ability to represent complex distributed conditions. We can reason along three different axes of a configuration: numeric state variables, topological configuration, and temporal progression. Topological restrictions allow us to model (in some abstract fashion) the configuration space of the nodes, so that error states related to the topology of neighboring nodes may be represented. Temporal modal operators can be used to represent sequences of states, a useful capability for debugging distributed finite state automata.

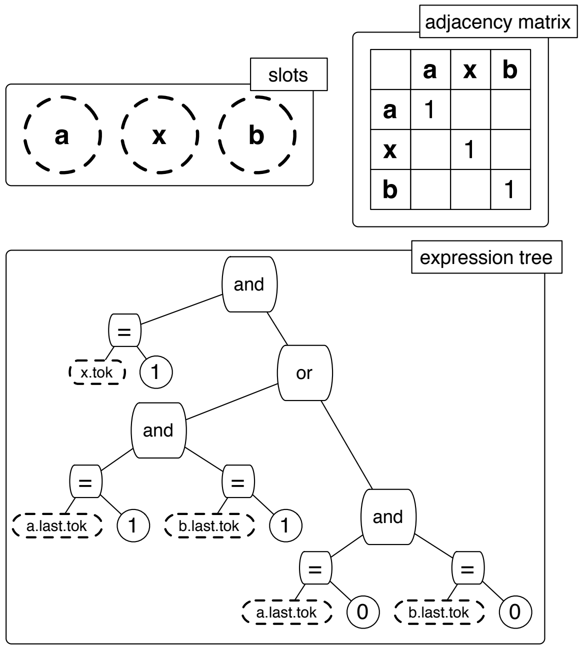


Figure 2. Example PatternMatcher object. Variables are shown with dotted outlines.

3.3 Detecting Errors

The main component in our distributed watchpoint implementation is the PatternMatcher object (Figure 2). A PatternMatcher consists of three subunits:

- an ordered list of n named slots that hold node id numbers
- an $n \times n$ bit adjacency matrix
- an expression tree that both represents the watchpoint expression and stores any accumulated state variables

A PatternMatcher may be empty (with none of its slots filled), partially filled (with some slots and state variables filled), or completely filled. A given PatternMatcher may be in one of three states: matched, failed, or indeterminate. The indeterminate state occurs when there is insufficient information in an empty or partially filled PatternMatcher to decide whether its expression is satisfied. Each PatternMatcher is a mobile data structure which represents one particular search to satisfy the watchpoint condition. A partially filled PatternMatcher is a search which is still in progress, where the empty slots are the incomplete portion of the search. A given node can host many PatternMatchers, representing various searches which are passing through the node, which collect state via the process described below.

3.3.1 Populating PatternMatcher Variables

In order to use PatternMatchers to determine whether watchpoint expressions have been satisfied, we *populate* the components of the PatternMatchers with information from different nodes. This process has three stages that occur at each timestep: i) filling the next available slot, ii) filling the expression tree, and iii) updating the adjacency matrix. When filling slots, we proceed in order: find the first slot that has not been assigned a value, then copy the local node's unique ID number to that slot variable. The name of the slot variable is then used in the second phase, where we traverse the expression tree and instantiate any references to the current slot's state variables with the values from the current node. Finally, we query the node for its current list of neighbors, and update the adjacency matrix with any connections between the current node and any nodes in previously filled slots. These steps are repeated for every node that the PatternMatcher acquires state information from.

As an example, we illustrate the action of the populating process on a simple watchpoint expression. We examine a single PatternMatcher as it is populated by two nodes in succession. The two nodes have unique id's 4 and 5, and gradient values of 12 and 10 respectively (Fig 3a). The PatternMatcher is first populated by node 4. This places the value 4 in slot `a`, and the value 12 in the variable `a.gradient`, as indicated by the arrows in (Fig 3c). The PatternMatcher

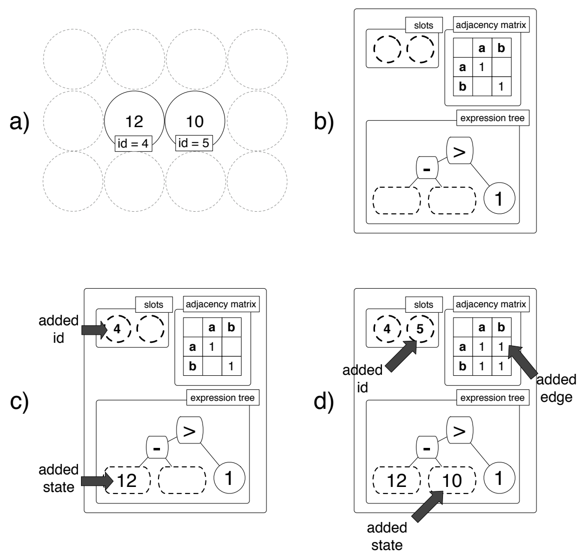


Figure 3. The population process, with arrows indicating where data is added. a) Node configuration, showing gradient value and node id. b) Initial (empty) PatternMatcher. c) PatternMatcher after being populated with node id 4. d) PatternMatcher after being populated with node id 5.

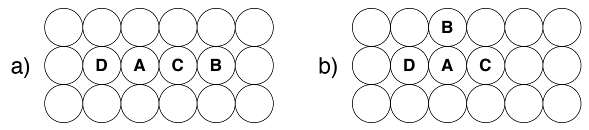


Figure 4. Unordered (a) and non-linear (b) subensembles for the set of nodes (A B C D).

is then transmitted to node 5, where its second slot is populated. This fills the slot `b` and the variable `b.gradient`. Additionally, as `a` is a neighbor of `b`, the relevant entries in the adjacency matrix are updated. The expression tree is now satisfied, and the PatternMatcher has matched the condition described by the watchpoint expression.

3.3.2 Centralized Algorithm

Our initial implementation relied on a single centralized procedure to update all PatternMatchers across an entire (simulated) ensemble. A central set of vectors maintained each node's state variables. At each timestep, the current values of all state variables used by active watchpoints were appended to the vectors, providing state history for the variables. The simulator also maintained a single set of PatternMatchers (S), which were updated and processed every timestep as described in Algorithm 1. This centralized algorithm is useful for simulated node ensembles, but debugging actual hardware required the development of a distributed technique for detecting watchpoint conditions.

3.3.3 The Need For Mutihop Messaging Within A Distributed Implementation

To detect the presence of distributed conditions without the presence of a central control entity, we use PatternMatchers as mobile state aggregators. PatternMatchers are transmitted from node to node via single-hop messaging, accumulating state until they fail or trigger. Unfortunately, using single-hop messaging, the subensemble topologies that can be detected are quite limited. Consider the two subensembles in Figure 4. Figure 4a illustrates a subensemble whose nodes lie in a linear path, but are not ordered sequentially. With only single-hop messaging, there is no way to propagate the slots (A B C D) in order. Figure 4b shows a subensemble which has an ordered path, if one is allowed to "backtrack" to previously visited nodes. In order to implement this backtracking (and thus detect this class of subensembles) we introduce multi-hop messaging and rerouting.

Multi-hop messaging is used to move a PatternMatcher back through previously visited nodes, so that it can continue to propagate from a different point in the subensemble. PatternMatchers that return to a node that they have already visited are said to have been *rerouted* to the node. Multi-hop messaging and rerouting allow us to detect subensembles of the kind seen in Figure 4b, which we call *nonlinear* ensembles, but not to detect those seen in Figure 4a, which are *unordered*. To detect unordered subensembles we would have to reorder the slots of the PatternMatcher to correspond to the ordering in the subensemble. To detect all such unordered configurations, we would have to search all permutations of the slots, a potentially expensive proposition (as a watchpoint of size four would have 24 such permutations).

3.3.4 Distributed Algorithm

The distributed implementation of our watchpoint system involves two components on each node: an update function and a message handler for incoming PatternMatchers. The update function is used at each timestep to create, populate, and spread PatternMatchers. Each node maintains three data structures:

- a set S of active PatternMatchers
- a set R of PatternMatchers to be rerouted to the node's neighbors
- a local history of watched state variables (used by the *populate* routine)

The PatternMatcher update function of the algorithm is executed at each timestep, and is composed of four phases. In the first phase, a new (empty) PatternMatcher is created, and added to the set S. Then for each PatternMatcher in S, its first open slot (and associated expression tree variables) is populated with information from the local node. If a PatternMatcher's expression tree is satisfied, the trigger action for the watchpoint is executed. If still indeterminate, the PatternMatcher is then spread via two types of messages: local messages sent to each of the node's neighbors, and multihop messages sent to all of the other nodes already identified as being in the PatternMatcher's slots. Every rerouted PatternMatcher in R is then sent via a local message to all of the node's neighbors. Finally, the sets S and R are cleared.

All messages used in the distributed algorithm carry five data fields:

- the PatternMatcher `p`
- a boolean flag `isMultihop`
- a boolean flag `possibleDup`
- the destination node `dest`
- and the source node `source`

We represent a complete message as <*p, isMultihop, possibleDup, source, dest*>. '$*$' is used to denote field values that we do not care about. Message handling is based on whether a given message is multihop or local. Local messages are handled by adding their PatternMatcher p to the local set S of active PatternMatchers.

Multihop messages are examined to see if their ultimate destination is the current node. If the message is in transit to another node, a breadth-first search from the current node to the destination is conducted within the adjacency matrix of the message's PatternMatcher. This provides the next step in routing the message to its destination, and the message is forwarded on to this next hop. If the message is meant for this node's neighbors, its PatternMatcher p is added to the local set R of rerouted PatternMatchers.

3.4 Operational Concerns

3.4.1 Machine Model

As mentioned in Section 3.2.1, the algorithms we have described assume a specific machine model: namely one that loops through three atomic phases: computation, communication, and state variable assignment. This abstract model is easily adapted to accommodate more general programming styles. We address the applicability of our machine model to three common styles: state-machine, event-handling, and run-loop.

The state-machine programming style (popular in embedded devices) treats the node as a finite-state automaton, which transitions to different states upon the reception of input signals. In this case, we can treat each state transition as a period of computation, followed by variable assignment (the new automata state, and any associated variables). The generation and reception of input signals are then the communications phase.

Event-handling systems are similar in nature to state-machines, as they respond to external interrupts with event handling routines. These systems lack the requirement of remaining within a transition system containing a finite number of states, and typically handle many more interrupts than a state-machine handles transitions. In this case, we can potentially treat each interrupt service request as a discrete timestep, with the interrupt serving as the communications phase. As this may impose an excessive burden on the communications infrastructure (it would imply sending out PatternMatchers after every interrupt is handled) we can group a number of closely spaced interrupt requests together to form a timestep. In this case, the interrupt handlers may access the same state variables, so one can either discard the intermediate values assigned within a timestep, or create additional intermediate state variables that can be used to track them.

Run-loop programming is the simplest of all programming styles to adapt for our purposes. We can treat each pass through the main loop as a timestep, and record only the variable values that are present at the end of each loop. If a program sends or receives messages multiple times within the run loop, we can arbitrarily designate one of those occurrences to be the "end" of the loop, and thus mark the point where variable values are recorded and PatternMatchers are propagated. Alternatively, we can make every instance of inter-node communication correspond to the end of a timestep, at which point a single pass through the run loop may contain multiple timesteps.

3.4.2 Snapshot Correctness

In any distributed algorithm which requires state from multiple asynchronously executing programs, there arises the question of consistency. Without some prior guarantees on the length of a timestep, or the presence of a shared clock, there is generally no way to obtain a "snapshot" of the state of multiple nodes that corresponds to one instance of wallclock time. What can be obtained is a *consistent* snapshot, or one that represents a set of states that could exist without any messages passed between the nodes. We examine the issue of snapshot coherence in two phases: in ordered subensembles (which do not require multihop messaging), and then in nonlinear subensembles.

In linear subensembles, the movement of a PatternMatcher from node to node serves as both the marker and state aggregator for a bounded-size Chandy-Lamport snapshot [8]. This allows us to capture a consistent snapshot of the nodes' state for any state variables whose values persist for at least one timestep.

In nonlinear subensembles, the problem becomes more complicated. As the multihop messaging phase of the algorithm introduces additional delays in state capture, it is no longer possible to ensure a consistent snapshot with the PatternMatchers alone. There are two possible solutions to this problem. If the timesteps of all nodes are known to be of equal length, then one can simply apply a backward temporal shift of 1 to all unfilled variables whenever a PatternMatcher moves as part of multihop routing. This ensures that the data the PatternMatcher collects when it reaches a new node has "aged" an amount equal to the number of multihop routing steps, which will be equal to the number of timesteps that have elapsed (by the assumption above, and the abstract machine model).

If there is no guarantee as to the length of timesteps on different nodes, the following approach may be used: when a PatternMatcher with m slots is created, it is given a timestamp by the local node. That timestamp is broadcast to all neighbors within m hops, which record the relevant state values and associate them with the timestamp. When a PatternMatcher arrives at a node, it requests the state variables associated with its timestamp. This is equivalent to separating the beacon and state aggregation phases of the Chandy-Lamport snapshot algorithm. The beacon is the timestamp broadcast, and the aggregation is performed by the rerouted PatternMatcher.

3.5 Efficiency

We also analyzed the overall execution time of the algorithms, and their scaling behavior as the size of the ensemble grows. In these tests, we used a 4-slot watchpoint expression, with the host program generating random variables that resulted in half of the active PatternMatchers being culled at each stage. Each node also ran a data aggregation and landmark routing program, to simulate a medium-intensity workload on the system. Tests were run on ensembles of various sizes, using the centralized and distributed implementations, as well as without any watchpoints enabled (for comparison).

Overhead for linear expressions is quite reasonable, with the centralized algorithm having an average overhead of 115% and the distributed algorithm an average overhead of 304%. The overhead for both algorithms is well within the range of other debugging tools like GDB [16] and Valgrind [30]. Unfortunately, the overhead for nonlinear expressions is much higher, being as high as 5182% in the worst case. While this may seem unacceptably high, we must consider the number of successful matches being found via the algorithm. With 1000 nodes, the nonlinear watchpoint finds 3.57 million matches over 100 timesteps. If we examine the time expended per match, we see that the time required as a function of the number of matches is actually quite low: less than one millisecond per match. If we assume that the system will be used to detect relatively infrequent events, then this level of performance is quite adequate.

To test this hypothesis, we compared a set of experiments on 1000 nodes, using a 4-slot watchpoint that discarded 99% of all active PatternMatchers after the first fill(). Results were quite encouraging, with an overhead of only 5%

(a 20x reduction) for linear subensembles detected via the centralized algorithm, and 43% (over a 100x reduction) for nonlinear subensembles detected with the distributed algorithm.

3.5.1 Optimizations

To reduce the storage, processing, and communications demands of our watchpoint system, we implement three optimizations: temporal span detection, early termination of candidate pattern matchers, and aggressive neighbor culling. These optimizations increase the watchpoint size and detection frequency that is feasible for a given ensemble of computational nodes.

3.5.2 Automatic Temporal Span Detection

For each state variable, we must determine the minimum amount of history that must be maintained by each node. We call this quantity the temporal span of the variable. Additionally, we must determine the minimum amount of total state (all state variables plus neighbor information) that must be maintained to allow for watchpoints that trigger in the future. This is the temporal extent of the system. To calculate the temporal span of a variable, we inspect each use of that variable in the watchpoint expression. For each use of the variable, we calculate the temporal extent by assigning a value of $+1$ to each `next`, and a value of -1 to each `last`. The sum is then the temporal extent for that particular use of the variable. The temporal span for the variable is the maximal difference between any two temporal extents. This is the amount of history which must be maintained for that variable, without factoring in multihop messaging delays. Similarly, the maximum positive extent over all variables specifies the size of the total state vector that must be maintained.

3.5.3 Early Termination

To reduce the number of active PatternMatchers, and thus the bandwidth and processing cost of the algorithm, we aggressively cull PatternMatchers that have no chance of succeeding. Before propagating a PatternMatcher to other nodes, we first check whether the expression tree can ever match. Just as with short-circuit evaluation of a boolean expression in programming languages like C, even if the PatternMatcher is not completely filled, subclauses of the expression tree may have already made the whole unsatisfiable. If this match can already be ruled out, the PatternMatcher is deleted, and no messaging and computation is required to check its descendants.

3.5.4 Neighbor Culling

We can reduce the set of neighbors to which a given PatternMatcher can spread by examining the topological constraints of the expression tree. If the constraint `neighbor(a b)` exists in the watchpoint, `b` is the next open slot, and `a` is already filled, then the PatternMatcher can only spread to neighbors of `a`. In the case of multiple topological restrictions, we generate a set of possible neighbors by traversing the tree from the bottom up, treating `and` as set intersection and `or` as set union. Without this optimization, we could conceivably spread a PatternMatcher to a large number of other nodes whose connectivity relationships would prevent the watchpoint expression from being satisfied.

4 Proposed Work

As stated in Section 1, the approach I plan to take during this thesis rests on the validation of two premises: that LDPs are useful in a variety of application domains, and that techniques for describing and detecting these LDPs are both theoretically and empirically sound. In the following sections I sketch a course of proposed research for each of these topics. The actual path of the research will, of course, be slightly different that this, as it will be informed by the difficulties encountered during the progress of the thesis.

4.1 Applicability to Distributed Application Domains

To establish the applicability of distributed predicate search to a wide variety of situations, I plan to demonstrate its use in three scenarios: distributed debugging, modular robotic motion planning, and hop-bounded service discovery. These three scenarios will also provide test cases for performance evaluation and (in the latter two cases) competing implementations to benchmark against.

The first application I plan to demonstrate is distributed debugging, specifically debugging the modular robots of the Claytronics [2] project. In my previous publications, I demonstrated a number of example watchpoints, and exposed the system's underlying dependence on host program behavior. I plan to reprise this analysis, exploring the effects of additional language features and optimizations on distributed watchpoint performance. Distributed debugging is an interesting test case, as it is one where successful matches are rare and search can terminate early a high percentage of the time. This means that many language features which would be too expensive for general programming usage would remain appropriate for use in distributed debugging.

The second application I plan to demonstrate is metamodule motion planning for Claytronics. As a follow-on to my early work on hole-based motion planning [14], Srinivasa and Dewey developed a general technique for modular robot motion planning (currently in review), which uses metamodules composed of multiple robots to transition a large ensemble from one 3D shape to another. I will demonstrate the core of this work, the detection of conditions for growth and deletion, implemented using distributed predicate search. This application will demonstrate performance under a large number of small (2 or 3 node) predicates.

Finally, I plan to demonstrate the use of distributed predicate search for hop-limited service discovery in generic multihop networks. As a concrete example, I plan to demonstrate the ability to locate high-bandwidth paths to a nearby gateway in RoofNet [5] or other MANET systems. This application requires reasoning about link-level variables and variably-sized routes.

Each of the three scenarios I plan to test stresses a different element of the LDP system: debugging requires ability to quickly discard large numbers of PatternMatchers, motion planning requires complex coordination and quick response while limiting message traffic, and service discovery requires the efficient detection of predicates that can span variably-sized subensembles. These disparate requirements provide a large problem space for LDP to tackle, and successfully addressing the needs of each will show the broad utility of the technique.

4.1.1 Syntactic Enhancements

To implement the proposed applications, the basic language described in my work on distributed watchpoints must be extended via additional language primitives. The exact set of primitives necessary for each application domain is dependent on many factors, but the proposed features below provide an example of the types of features that would be considered.

Edge Variables

A straightforward extension to the availability of state variables on nodes is to make them available to edges. The exact set of edge variables would be application dependent, but one can easily see the utility of being able to represent available bandwidth, distance or bearing, and signal-to-noise ration for links between nodes.

Set Variables

The current watchpoint description language supports only scalar types: booleans and floating points. The addition of set (or array) variables would allow for the representation and querying of to-many relationships. In addition to including sets as a basic datatype, the language would also have to be modified to support common set operations, such as size(),union(), and subset(). This functionality allows for reasoning about sets of neighbors in a distributed system, and would provide several interesting optimization opportunities, as it is obviously disadvantageous to transmit entire sets during the distributed search process.

Named Searches

The ability to name searches, and then query a node for its membership status with regard to a particular query, exposes the capability to perform hierarchical search. As an example, one could define a low-level search predicate that would locate logical groups in an ensemble, and then use a higher-level search to express properties in terms of those groups, rather than their underlying nodes.

Quantified Kleene-$*$

One glaring limitation in the current distributed watchpoint system is its inability to represent conditions that involve a variable number of nodes. To provide this functionality, I plan to implement a version of the Kleene-$*$, allowing one or more named nodes to be replicated inside the search expression. In order to ensure that the scope of the resulting search remains bounded, I plan to require that the replication be bounded within minimum and maximum values. This addition would imply the modification of all other comparison operations, so that one could express properties that exist between these replicated nodes.

Count() operation

A relatively simple addition, count() takes one or more comma-separated boolean expressions and returns the number of those expressions that evaluate to true. Count() is interesting for three reasons: (i) it is connected to the use of higher-order logics, as it can upconvert from boolean back to numeric values, (ii) count(), when combined with quantified Kleene-$*$, allows for the concise expression of complicated compound properties, and (iii) count(), when used as one half of a comparison operator, lends itself to aggressive optimization via short-circuit evaluation.

Interval-Based Sampling

The final proposed enhancement to the language involves not only the syntax of the language, but also the method by which state variable data is gathered. The current model of state capture assumes that there is some interval, or "tick", when all state is captured. This tick-based sampling will miss any transitions in state variables that occur between sample times, preventing the capture of transient conditions, and introducing a source of race conditions. An alternate model would be edge-triggered sampling, where a variable is sampled every time it changes state. While this would capture all events that occurred in the system, it mave produce a large quantity of state, which may be untenably large for transmission or storage purposes. As a compromise, I plan to implement set-based storage of variable values. For a particular timestep, every variable maintains a set of the discrete values it assumed during that tick. By placing limits on the frequency of timesteps, and the maximum size of the set, this system can emulate both tick-based sampling and edge-triggered detection. The introduction of set-based variables would necessitate changing the semantics of all mathematical and comparison operators in the system. The extension of mathematical operators is relatively straightforward, but the extension of comparators reveals two different semantic interpretations: that a comparison between two sets is always true (for any pair of elements), or that it is true for some pair of elements. To expose this semantic difference, I plan to modify the language to allow for the specification (on a per-comparison basis) of which mode to use.

4.2 Theoretical Validation and Experimental Evaluation/Improvement

The second branch of my work rests on the validation of LDPs and the predicate search system as a valid technique, from both a formal and an experimental standpoint.

4.2.1 Formal Properties

To place my work on a more sound formal footing, I plan to analyze the effect of the various language constructs and algorithms on the proof properties of the predicates. This will involve the development of a formal grammar, type system, and operational semantics for the predicates, in either a process calculus or logic fragment. The choice of formal model is important, as the underlying properties of the calculus or logic will affect the set of properties that are easy to prove. Once I have developed the formal encoding of the LDP system, I will need to validate it by proving basic progress and safety theorems. From there, I will use the formal framework to evaluate which language features provide guarantees that a predicate is possibly true or definitely true, under various classes of distributed systems. I anticipate that there will be several language features that will not be provably consistent for general distributed systems, and that these features will work only in a restricted subset of distributed ensembles (those with bounded message delay, or perhaps shared clocks).

4.2.2 Performance Evaluation

To validate the performance of the predicate search system, I will evaluate the computation and communication costs of the LDP implementations of distributed watchpoints, motion planning, and resource discovery. I plan to evaluate both the total cost and the cost per node, as well as the cost per match. To get a more nuanced view of the system's cost, I will evaluate multiple topologies, ensemble sizes, predicate sizes, and match probabilities. Finally, I will compare the performance of the LDP motion planner and resource discovery implementations with the same algorithms implemented in lower-level programming languages (C++, Java, etc.).

4.2.3 Search Strategies & Performance Enhancements

The next stage of the work will focus on enhancing the performance of the predicate search system. This section is, of necessity, slightly more tentative than others, as the necessity and efficacy of various optimizations is difficult to predict at this time. The optimizations that I plan to test can be broadly divided into three categories: optimizations of messaging volume, optimizations of search strategy, and optimizations in PatternMatcher short-circuiting.

As network traffic is often the most expensive commodity in a distributed system, any optimizations that reduce extraneous messages could potentially provide a efficiency boost. The simplest optimization for reducing network traffic that I plan to implement is batching messages. Instead of sending each individual PatternMatcher as its own packet, the predicate search system would batch together multiple PatternMatchers with the same destination. This would amortize the per-packet transmission overhead over multiple PatternMatchers, but introduce an additional source of latency. Another optimization to reduce message volume would be automatic culling of filled subexpression trees. To perform a distributed state comparison, the PatternMatcher must store and transmit two or more 32-bit data values. From the perspective of the search system, these values are unnecessary, as we care only about the high order bit from the comparison operator. Culling subtrees and replacing them with the results of the top-level comparison could significantly reduce message size. Finally, we could take the additional step of caching remote state variables as they pass through a node, allowing for certain remote state comparisons to take place purely locally. This optimization is not compatible with subtree culling, and could potentially expose the PatternMatcher to out of date or incorrect information, so its implementation would require careful consideration.

The basic search algorithms developed for [15] and [13] allow for the detection of linear subensembles, as well as ordered subensembles with backtracking. An obvious extension would be to develop an algorithm for non-ordered subensembles which would enable detection of a larger class of properties. Allowing non-ordered subensembles would, however, greatly increase the number of possible subensembles, and thus the cost of the predicate search algorithm. Another possible optimization would be to replace the PatternMatcher-based search with some other technique, possibly an automata-derived method such as those used in regular path queries [6]. Finally, I plan to consider the problem of success notification, in particular the potential need to efficiently inform multiple nodes of one or more successful matches. Efficient notification will likely require some form of multicast spanning tree [37] or other overlay construct.

Finally, I plan to investigate the additional opportunities for short-circuiting that the additional language constructs introduce. As an example, if we were presented with an expression of the form `count(...)>1`, where the body of the count operation contained 3 or more subexpressions, it would be unnecessary to evaluate more than two of them if the two subexpressions first evaluated were both true.

4.2.4 Fault Tolerance

For LDPs to be generally useful in real distributed applications, the underlying search algorithm must be robust to a certain level of failures, or at least provide the programmer with notification of such failures. I plan to explore the effects of non-correlated, non-Byzantine node and communications failures on the operation of the LDP search algorithm. This will involve three tasks: observing the behavior of the base algorithm under failures, implementing failure detection mechanisms, and finally implementing failure recovery or avoidance. On first inspection, the non-multihop search algorithm appears to be relatively robust to failures in communication, as such failures will merely prevent subensembles containing the failed node from matching. To prevent transient communications errors from removing these nodes from the search space, I plan to explore the possibility of caching state data at remote nodes (as mentioned in the optimization section above), so that a node can be searched even when temporarily unavailable. Multihop messaging and backtracking for action execution are more vulnerable to failure, as a failure anywhere in the matching subensemble could conceivably prevent the multihop messages from reaching their destination. Preventing this occurrence will likely require the implementation of a robust routing algorithm instead of the simple path backtracking currently used.

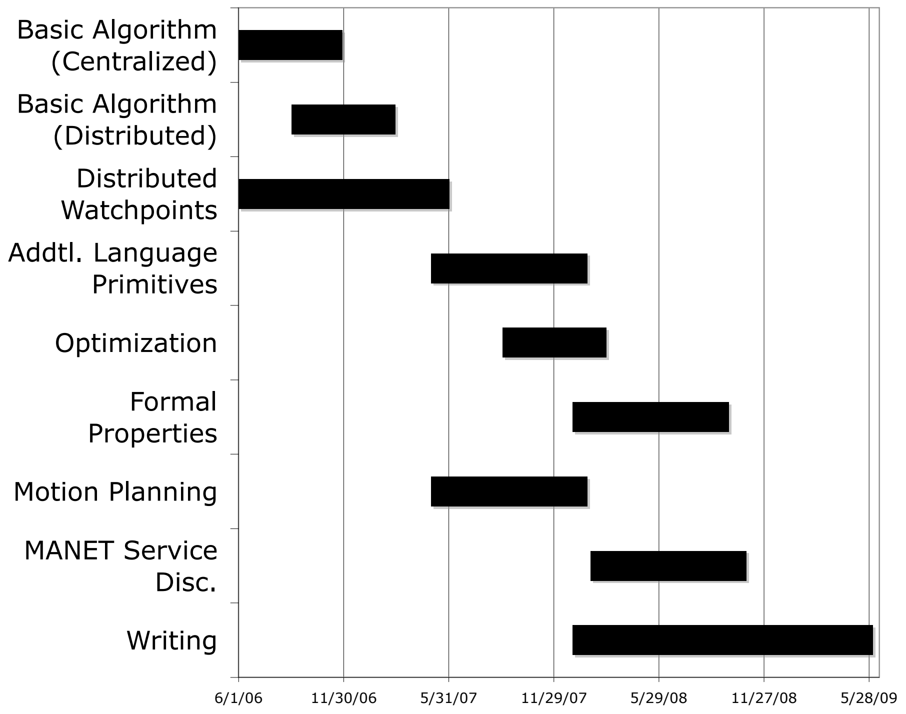
5 Timeline

Figure 5 shows my proposed schedule to complete the thesis. Figure 5(a) depicts the global picture of the timeline for all the thesis work. I have already finished the first three pieces of work, namely, basic algorithms for both centralized and distributed predicate detection, and proof-of-concept implementation of distributed watchpoints. I plan to spend 9 months working on additional language primitives, 6 months working on performance optimization, 9 months working on characterizing the formal properties of the system, and 18 months writing the thesis.

References

[1] Etnus TotalView. [Online]. Available: http://www.etnus.com/

[2] B. Aksak, P. S. Bhat, J. Campbell, M. DeRosa, S. Funiak, P. B. Gibbons, S. C. Goldstein, C. Guestrin, A. Gupta, C. Helfrich, J. Hoburg, B. Kirby, J. Kuffner, P. Lee, T. C. Mowry, P. S. Pillai, R. Ravichandran, B. D. Rister,



(a) Timeline for all the thesis work

Figure 5. Thesis timeline

S. Seshan, M. Sitti, and H. Yu, "Claytronics: highly scalable communications, sensing, and actuation networks [demo abstract]," in *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM Press, 2005, pp. 299–299.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[4] M. Ashley-Rollman, S. Goldstein, P. Lee, T. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *Proceedings of the IEEE International Conference on Robots and Systems IROS '07*, 2007.

[5] S. Biswas and R. Morris, "Opportunistic routing in multi-hop wireless networks," in *Proceedings of the ACM SIGCOMM '05 Conference*, Philadelphia, Pennsylvania, August 2005.

[6] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi, "Reasoning on regular path queries," in *Proceedings of SIGMOD*, 2003. [Online]. Available: citeseer.ist.psu.edu/article/calvanese03reasoning.html

[7] S. Carr, J. Mayo, and C.-K. Shene, "Race conditions: A case study," *The Journal of Computing in Small Colleges*, vol. 17, no. 1, pp. 88–102, October 2001.

[8] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, February 1985.

[9] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier, "Local and temporal predicates in distributed systems," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 1, pp. 157–179, 1995.

[10] C. M. Chase and V. K. Garg, "Detection of global predicates: Techniques and their limitations," *Distributed Computing*, vol. 11, no. 4, pp. 191–201, 1998. [Online]. Available: citeseer.ist.psu.edu/article/chase98detection.html

[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[12] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, vol. 26, no. 12, 1991, pp. 167–174. [Online]. Available: citeseer.ist.psu.edu/cooper91consistent.html

[13] M. De Rosa, S. C. Goldstein, P. Lee, J. Campbell, and P. Pillai, "Distributed watchpoints: Debugging large modular robotic systems (in preparation)," *International Journal of Robotics Research*, Special Issue on Modular Robotics 2007.

[14] M. DeRosa, S. Goldstein, P. Lee, J. Campbell, and P. Pillai, "Scalable shape sculpting via hole motion: Motion planning in lattice constrained modular robots," in *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '06*, 2006.

[15] M. DeRosa, S. Goldstein, P.Lee, J. Campbell, and P. Pillai, "Distributed watchpoints: Debugging large multi-robot systems," in *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '07*, 2007.

[16] FSF, *GDB: The GNU Project Debugger*, Free Software Foundation, 2007. [Online]. Available: http://www.gnu.org/software/gdb/

[17] V. K. Garg and B. Waldecker, "Detection of weak unstable predicates in distributed programs," in *Global States and Time in Distributed Systems*, Z. Yang and T. A. Marsland, Eds. IEEE Computer Society Press, 1994. [Online]. Available: citeseer.ist.psu.edu/garg94detection.html

[18] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[19] L. Huang, X. Nguyen, M. Garofalakis, J. Hellerstein, A. D. Joseph, M. Jordan, and N. Taft, "Communication-efficient online detection of network-wide anomalies," in *26th Annual IEEE Conference on Computer Communications (INFOCOM'07)*, 2007.

[20] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal, “Efficient distributed detection of conjunctions of local predicates,” *Software Engineering*, vol. 24, no. 8, pp. 664–677, 1998. [Online]. Available: citeseer.ist.psu.edu/hurfin96efficient.html

[21] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall, “A wakeup call for internet monitoring systems: The case for distributed triggers,” in *Proc. 3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

[22] B. Karp and H. T. Kung, “Greedy perimeter state routing for wireless networks,” in *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, Boston, Massachusetts, August 2000, pp. 243–254.

[23] K. B. Lamine and L. Kabanza, “Reasoning about robot actions: A model checking approach,” *Advances in Plan-Based Control of Robotic Agents*, pp. 123–139, 2002. [Online]. Available: citeseer.ist.psu.edu/ben02reasoning.html

[24] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[25] ——, “”Sometime” is sometimes ”not never”: on the temporal logic of programs,” in *POPL ’80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1980, pp. 174–185.

[26] S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande, *Computational Genomics*. Horizon Press, 2002, ch. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology.

[27] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, “Implementing declarative overlays,” in *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.

[28] F. Mattern, “Virtual time and global states of distributed systems,” in *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*. Elsevier, 1989. [Online]. Available: citeseer.ist.psu.edu/mattern89virtual.html

[29] R. Milner, “The polyadic pi-calculus: A tutorial,” The University of Edinburgh, Tech. Rep. ECS-LFCS-91-180, October 1991. [Online]. Available: http://www.lfcs.inf.ed.ac.uk/reports/91/ECS-LFCS-91-180/

[30] N. Nethercote and J. Seward, “Valgrind: A program supervision framework,” *Electronic Notes in Theoretical Computer Science*, 2003.

[31] R. Newton, G. Morrisett, and M. Welsh, “The regiment macroprogramming system,” in *IPSN ’07: Proceedings of the 6th international conference on Information processing in sensor networks*. New York, NY, USA: ACM Press, 2007, pp. 489–498.

[32] R. Obermarck, “Distributed deadlock detection algorithm,” *ACM Trans. Database Syst.*, vol. 7, no. 2, pp. 187–208, 1982.

[33] I. Phillips and M. Vigliotti, “On reduction semantics for the push and pull ambient calculus,” in *Proceedings of IFIP International Conference on Theoretical Computer Science (TCS 2002), IFIP 17th World Computer Congress, August 2002, Montreal*. Kluwer, 2002, pp. 550–562. [Online]. Available: http://www.doc.ic.ac.uk/ iccp/papers/

[34] A. Pnueli, “The temporal logic of programs,” in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977, pp. 46–67.

[35] R. Schwarz, “Causality in distributed systems,” in *EW 5: Proceedings of the 5th workshop on ACM SIGOPS European workshop*. New York, NY, USA: ACM Press, 1992, pp. 1–5.

[36] G. Tel and F. Mattern, “The derivation of distributed termination detection algorithms from garbage collection schemes,” in *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 1. ACM Press, January 1993, pp. 1–35. [Online]. Available: citeseer.ist.psu.edu/article/tel93derivation.html

[37] D. Waitzman, C. Partridge, and S. Deering, “Distance Vector Multicast Routing Protocol,” RFC 1075 (Experimental), Nov. 1988. [Online]. Available: http://www.ietf.org/rfc/rfc1075.txt

Appendix:Algorithms

Algorithm 1 Centralized Watchpoint Update

```
$S = \emptyset$
for all modules $m$ do
  create new PatternMatcher $p$ from the watchpoint text
  populate $p$ with $m$
  $S = S \cup p$
end for
while $S \neq \emptyset$ do
  $T = \emptyset$
  for all PatternMatchers $p \in S$ do
    if $p$ matches then
      execute trigger action for $p$
    else if $p$ is indeterminate then
      for all neighbors $n$ of modules in $p$'s slots do
        $p_1$ = clone($p$)
        populate $p_1$ with $n$
        $T = T \cup p_1$
      end for
    else
      #failure: PatternMatcher is ignored
    end if
    $S = S - p$
  end for
  $S = T$
end while
```

Algorithm 2 Distributed Algorithm: Message Handler

```
message $m$ received by module $n$
if $m.isMultihop$ then
  if $m.dest = n$ then
    $n.R = n.R \cup m.p$
  else
    $r$ = nextRoutingStep($n$,$m.dest$,$m.p$)
    $m.source$ = $n$
    send $m$ to $r$
  end if
else
  if !$m.possibleDup$ or !isDuplicate($m, n$) then
    $n.S = n.S \cup m.p$
  end if
end if

isDuplicate(message $m$,module $n$)
  $i_s$ = index of $m.s$ in $m.p$'s slots
  $i_{max}$ =max index of $n$'s neighbors in $m.p$'s slots
  return $i_s \geq i_{max}$

nextRoutingStep(module $s$,module $d$,PatternMatcher $p$)
  calculate BFS over $p$'s adjacency matrix (from $s$)
  $r$ = minimal path from $s$ to $d$ (using BFS distances)
  return the first element of $r$
```

Algorithm 3 Distributed Algorithm: PatternMatcher update

```
on each module $n$
create new PatternMatcher $p'$ from the watchpoint text
$S = S \cup p'$
for all PatternMatchers $p_1 \in S$ do
  populate $p_1$ with $n$
  if $p$ matches then
    execute trigger action for $p_1$
  else if $p_1$ is indeterminate then
    message $m_1$ = $<$ $p_1$,false,*,*,*$>$
    send $m_1$ to each of $n$'s neighbors
    for all modules $d$ in $p_1$'s slots s.t. $d \neq n$ do
      message $m_2$ =$<$ $p_1$,true,*,$n$, $d$ $>$
      module $r$ =nextRoutingStep($n$,$d$,$p_1$)
      send $m_2$ to $r$
    end for
  else
    #failure: PatternMatcher is ignored
  end if
end for
for all PatternMatchers $p_2 \in R$ do
  message $m$ = $<$ $p_2$,false,true,*,*$>$
  send $m$ to each of $n$'s neighbors
end for
$S = \emptyset, R = \emptyset$
```