

# *forward*

*a future of reliable wireless ad hoc networks of roaming devices*

## A CSP Framework for Analysing Fault-Tolerant Distributed Systems

June 30, 2004

**Record of Changes**

Date	Version	Comment
30-06-2004	1.0	First Issue

**Authorisation**

Dr. Sadie Creese  
FORWARD Steering Committee Member

Date

## **Authors**

William Simmonds, QinetiQ, wfsimmonds@qinetiq.com

Tim Hawkins, QinetiQ, thawkins@qinetiq.com

## Executive summary

This document reports the second of the four groups of studies (constituting Workpackage 5 of the FORWARD project) aimed at establishing basic mechanisms for assuring quality of service in ad-hoc networks, a core component of Next Wave, future ubiquitous computing, environments. The first two studies of Workpackage 5 address black-and-white questions of “correctness”; the first of these focussed on the topic of routing, while this document addresses distributed data replication.

Industry has traditionally employed simulation and testing in its verification of computer systems. Whilst these approaches are proven to identify errors/bugs, they are not exhaustive; unless you have simulated every situation, or tested every configuration, you simply cannot be sure that your system will always behave correctly. In contrast, formal verification techniques can offer exhaustive analysis (equivalent to 100% coverage testing of system states) which has often not only led to the discovery of errors previously undetected, but also, in the absence of errors, provides a guarantee of correctness. The formal verification of systems of realistic size and complexity in itself requires the help of computers. We have chosen to use CSP and the accompanying model checker FDR to enable formal verification of distributed data replication-based systems.

Our methodology uses the fault-tolerance library, a CSP framework and tool suite for helping in the design, modelling and verification of fault-tolerant distributed systems. We demonstrate the use of the method by reference to a selection of algorithms used by OceanStore. OceanStore is an architecture for a global-scale, persistent, distributed storage mechanism. Some correctness results are obtained for these algorithms for small static networks.

Finally the method of CSP structural induction is introduced; this technique can enable correctness results of algorithms to be proved for arbitrary large networks. Possibilities for support for structural induction within the fault-tolerance library are described.

A software demonstrator of the OceanStore models is available from the project website at [www.forward-project.org.uk](http://www.forward-project.org.uk).

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context . . . . .	1
1.2	Replication and Quality of Service . . . . .	1
1.3	The Fault-Tolerance Library and CSP/FDR . . . . .	1
1.4	OceanStore . . . . .	3
1.5	Structure of Document . . . . .	3
<b>2</b>	<b>The Fault-Tolerance Library</b>	<b>4</b>
2.1	Introduction . . . . .	4
2.2	The Graphical Interface . . . . .	4
2.3	State-Machine Language . . . . .	6
2.4	Templates and Core Processes . . . . .	7
2.5	Specification of System Properties . . . . .	7
2.6	Model Checking of Properties for Networks of Fixed Size . . . . .	8
2.7	Structural Induction . . . . .	10
<b>3</b>	<b>A Case Study: OceanStore</b>	<b>11</b>
3.1	Overview of OceanStore . . . . .	11
3.2	Probabilistic Data Location and Routing . . . . .	11
3.3	Modelling . . . . .	14
3.4	Results for Networks of Fixed Size . . . . .	16
<b>4</b>	<b>Structural Induction in CSP: Possibilities for FT Library Support and Application to OceanStore</b>	<b>18</b>
4.1	Rationale . . . . .	18
4.2	Support for CSP Structural Induction of OceanStore . . . . .	18
4.3	Other Future Work . . . . .	21
<b>5</b>	<b>Conclusions</b>	<b>24</b>
<b>A</b>	<b>OceanStore State Machine Scripts</b>	<b>25</b>
A.1	Update Algorithm . . . . .	25
A.2	Query Algorithm . . . . .	27
	<b>References</b>	<b>30</b>

**This page is intentionally blank**

# 1 Introduction

## 1.1 Context

This document reports the second of the four groups of studies (constituting Workpackage 5 of the FORWARD project) aimed at establishing basic mechanisms for assuring quality of service in ad-hoc networks, a core component of Next Wave, future ubiquitous computing, environments. The first two studies of Workpackage 5 address black-and-white questions of “correctness”; the first of these focussed on the topic of routing, while this document addresses distributed data replication.

Industry has traditionally employed simulation and testing in its verification of computer systems. Whilst these approaches are proven to identify errors/bugs, they are not exhaustive; unless you have simulated every situation, or tested every configuration, you simply cannot be sure that your system will always behave correctly. In contrast, formal verification techniques can offer exhaustive analysis (equivalent to 100% coverage testing of system states) which has often not only led to the discovery of errors previously undetected, but also, in the absence of errors, provides a guarantee of correctness. The formal verification of systems of realistic size and complexity in itself requires the help of computers. We have chosen to use CSP and the accompanying model checker FDR to enable formal verification of distributed data replication-based systems.

## 1.2 Replication and Quality of Service

Replication is a fundamental technique that is well established in computer hardware, operating systems, distributed databases, and distributed file systems design. Network replication involves the storing of multiple copies of data objects in distributed locations throughout the network. Accesses to data are satisfied by copies stored nearby, thus saving the need to route requests all the way back to the original source. This results in four significant benefits:

- reduced access latency,
- reduced bandwidth consumption,
- server load balancing, and
- improved data availability/redundancy.

From a quality-of-service (QoS) perspective, there is an important distinction between caching and replication. Caches have relatively small storage capacity, and therefore have to evict old objects to make room for new ones; caches cannot therefore provide any guarantees of data persistence. Replication, on the other hand, represents a service commitment to keep a persistent copy of the object.

A distributed data replication system should therefore provide a method of data storage that is guaranteed to a very high degree of certainty to be resilient to loss or destruction of individual servers. Information stored on a such a system must be extremely *durable*. Furthermore, archiving of information should be automatic and reliable.

## 1.3 The Fault-Tolerance Library and CSP/FDR

The fault-tolerance library (FT Library) is a CSPm [9] framework and tool suite for helping in the design, modelling and verification of fault-tolerant distributed systems. Models constructed using the facilities provided by the fault-tolerance library can be automatically checked using the FDR refinement checker [8].

The FT Library allows one to specify, amongst other things, the replication of components. Replication is an important means of achieving greater dependability - the more nodes in the system, the more tolerant the system can be made to the failure of some of those nodes. An example of the

use of replication is OceanStore, where data may be replicated over a number of nodes. Section 2 of this report includes a detailed description of the fault-tolerance library.

CSP is a process algebra which is useful for describing systems that interact by communication. A system is modelled as a *process* (itself possibly constructed from a collection of processes) that interacts with its environment by means of atomic *events*. Communication is synchronous; an event takes place precisely when both the process and the environment agree on its occurrence. The syntax of CSP provides a variety of operators for modelling processes, and the associated algebra provides rewrite laws. Primitive operators include process prefix, sequential composition, deterministic and non-deterministic choice, parallelism, interleaving, hiding, recursion, deadlock and successful termination:

$$P ::= a \rightarrow P \mid P \S P \mid P \square P \mid P \sqcap P \mid P \parallel P \mid P \parallel\!\!\parallel P \mid \\ P \setminus b \mid \mu X \bullet F(X) \mid STOP \mid SKIP$$

The collection of mathematical models and associated semantics that make up CSP facilitate the capture of a wide range of process behaviours. The traces model captures the observable traces of events that a CSP process might exhibit. The failures model captures not only the traces of a process, but also those events it can refuse to perform after a particular trace. The failures divergences model also captures information about events on which a process may diverge (perform infinitely often) from a particular state.

The theory of refinement in CSP allows correctness conditions to be encoded as refinement checks between processes. If process  $P$  refines process  $Q$ , then all of the possible behaviours of  $P$  must also be possible behaviours of  $Q$  (although  $Q$  may also possess many other behaviours). Therefore,  $P$  is a correct, and more deterministic, implementation of  $Q$ . This notion of refinement holds for all three of the semantic models, where the possible behaviours of processes are interpreted in terms of the semantic model under consideration.

Refinement is transitive, so if process  $R$  is refined by process  $S$  (written  $R \sqsubseteq S$ ) and  $S$  is refined by  $T$  then  $R$  is refined by  $T$ :

$$R \sqsubseteq S \wedge S \sqsubseteq T \Rightarrow R \sqsubseteq T$$

All CSP operators are monotonic with respect to refinement:

$$R \sqsubseteq T \Rightarrow C[R] \sqsubseteq C[T]$$

where  $C[\ ]$  is a context built from CSP operators and constants. This facilitates compositional development of systems. Imagine we want to prove:

$$Spec \sqsubseteq C[System]$$

then we can break this into two parts. First, find a process that does refine the desired specification:

$$Spec \sqsubseteq C[P]$$

and second, prove that:

$$P \sqsubseteq System$$

Then, by monotonicity:

$$C[P] \sqsubseteq C[System]$$

and by transitivity:

$$Spec \sqsubseteq C[P] \wedge C[P] \sqsubseteq C[System] \Rightarrow Spec \sqsubseteq C[System]$$

as required.

The FDR tool takes CSPm, a machine-readable dialect of CSP, as its input syntax, and can be used to check not only refinement but also determinism, deadlock-freedom and livelock-freedom of processes. CSPm is the combination of a rich data language, based on functional programming, and the CSP process algebra. See [9] for a detailed description of CSPm.

## 1.4 OceanStore

Our description of the fault-tolerance library is illustrated through using the library to model and analyse elements of OceanStore [6].

OceanStore is an architecture for a global-scale, persistent, distributed storage mechanism. It was designed with two goals in mind. The first of these is that it can be constructed from an entirely untrusted infrastructure, and that it is therefore resilient to server crashes and information leakage to third parties. In order that such an infrastructure can be used without data being compromised, all data is protected through redundancy and cryptographic techniques.

The second goal is that OceanStore supports *nomadic data*. Nomadic data is defined to be data that is allowed to flow freely. In a system as large as OceanStore, locality of data is of extreme importance; therefore, a stated aim is that data may be cached *anywhere, anytime*.

The elements of OceanStore that we have modelled using the fault-tolerance library are related to data location and routing. The mechanism that performs these processes is fairly sophisticated, as a result of the fact that objects in the OceanStore are free to reside at *any* of the servers. This fact allows flexibility in selecting policies for replication, availability, caching, etc., but has the side-effect that the process of locating these objects is rather complicated.

The mechanism used is a two-tiered approach, with a fast probabilistic algorithm backed up by a slower, reliable hierarchical method. The reason for this is that objects that are accessed frequently are likely to be located near to where they are being used. The probabilistic algorithm may route to objects rapidly if they reside nearby, but success is not guaranteed. So if this attempt fails, a large-scale hierarchical data structure locates objects wherever they are in the OceanStore.

The focus of our attention so far has been the probabilistic data location and routing algorithm.

## 1.5 Structure of Document

In Section 2 we give an overview of the fault-tolerance library. In Section 3 we describe our modelling of OceanStore's data location and routing algorithms within the fault-tolerance library framework for a limited topology. In Section 4 we discuss the possible application of CSP structural induction to our OceanStore models and consider the possibilities for future support within the context of the fault-tolerance library. In Section 5 we make our conclusions.

## 2 The Fault-Tolerance Library

### 2.1 Introduction

The fault-tolerance library (FT-Lib) is a tool suite for helping in the design, modelling and verification of distributed systems. It adheres to the *message-passing* model of communication - i.e. intra-node communication is by way of message-passing, as opposed to *shared-variables*. The FT-Lib also adopts the *assumption-commitment* paradigm for the verification of systems.

Work on the fault-tolerance library began under the Assessing Dependable Distributed Systems CRP; the library has been extended within FORWARD to support the OceanStore analysis.

The main elements of the library are as follows.

- A 'user-friendly' graphical interface for describing the high-level view of the distributed system - essentially the system's communications topology.
- A 'user-friendly' IO state-machine language that is used to describe the processing of the individual, non composable nodes of the system.
- A library of *templates* of dependability mechanisms, such as N-modular redundancy, standby-systems and self-correcting mechanisms, expressed in the FT-Lib's graphical notation.
- A library of dependable 'core processes', such as voting mechanisms, load-balancers, simple databases and distributed functions, written in the state-machine syntax or in particular modelling languages supported by the library.
- Translators that map graphs and state-machines expressed in the FT-Lib syntax to compilable models in specific modelling or theorem-proving languages. These models may then be checked using commercially available model-checkers or theorem-provers. Currently only Communicating Sequential Processes (CSP) translators are provided, but it is envisaged that future translators will include Z and Prism.

In the following sections of this document, we shall discuss the above elements in more detail.

### 2.2 The Graphical Interface

The graphical interface tool is currently provided by way of the graph-drawing package Dia [1]. Dia allows users to draw named diagrams, or 'graphs' of nodes connected by lines. Nodes can come in any of a number of shapes selected from pre-defined Dia palettes. Nodes can be connected to other nodes by different types of lines selected from pre-defined palettes.

A special Dia palette has been created for the FT-Lib; the shapes in the palette are as in Figure 1. All FT-Lib palette shapes are annotated by text.

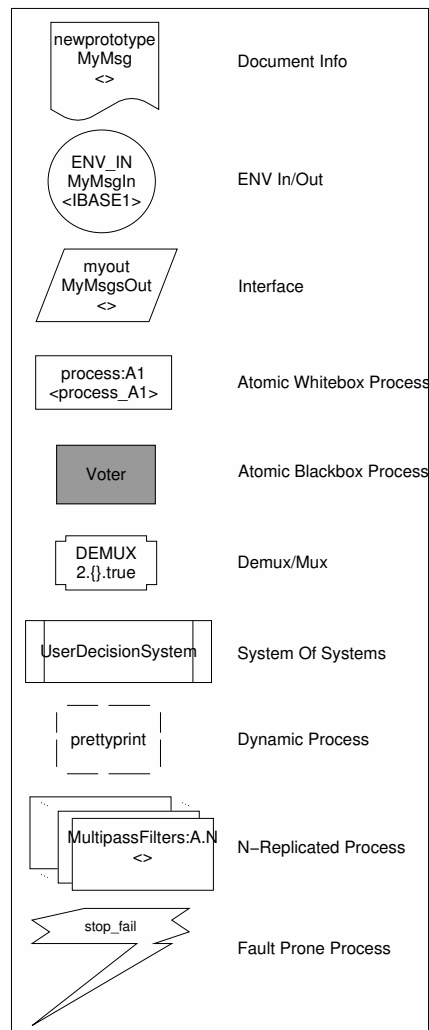


Figure 1: The FT-Lib Dia Palette

The only lines used in FT-Lib Dia graphs are one-way (uni-directional) arrows. An arrow going into a node represents an *input port* of that node, and an arrow coming out of a box represents an *output port* of that node.

All FT-Lib Dia graphs are *connected graphs* - i.e. all nodes in the graph are connected (by an arrow) to at least one other node.

The semantics attributed to the FT-Lib palette shapes are as follows.

- A 'rectangle' represents a non-decomposable user-defined process. These shapes are annotated with text citing the name and *instance identifier* of the process - e.g. 'VOTER\_A' declares an instance 'A' of some process called 'VOTER'.
- A 'tripartite rectangle' represents a *sub-system*, the sub-system itself being defined by an FT-Lib Dia graph.<sup>1</sup> In effect it is a 'placeholder' for another graph.
- A 'trapezium' and a circle respectively represent *user* and *environmental* interfaces. One may think of user interfaces as pertaining to users local to some process, while the environmental interfaces represent the communications interface of the whole system with other (perhaps

<sup>1</sup> Graphs can be nested, but not recursive - i.e. a named graph cannot appear as a sub-system of one of its own cited sub-systems.

unspecified) systems. There is at most one ENV\_IN and one ENV\_OUT circle. A graph is said to be *open* if it has both ENV circles, *open-closed* if it has an ENV\_IN but no ENV\_OUT, *closed-open* if it has an ENV\_OUT but no ENV\_IN, and *closed* if it has neither ENV circle. If a graph is intended to be cited as a sub-system (as above) of another graph, then it should not be closed.

- A 'dashed' rectangle represents a *dynamic* (or *transient*) process.
- The 'stacked rectangles' represent N-replicated components. The text in a stacked rectangle shape specifies the process that is being replicated and the number of replicates - e.g. 'N PROCESS\_A' indicates N replicates of some process, 'PROCESS\_A'.
- A rectangle with 'corners cut out' represents a *connector box*. There are four types of connector boxes, namely *multiplexers*, *de-multiplexers*, simple *uni-directional* and simple *bi-directional*.
- A 'lightning strike' dialogue box connected to some other box, B, say, indicates that B is 'fail-prone' in some way. The text in the lightning strike box indicates the way in which the box is fail-prone. A box can exhibit four types of failure, namely *sporadic stop-failure*, *sporadic Byzantine failure*, *sporadic corruption*, and *timing failure*. The *frequency* with which the node is prone to such failures can be specified as part of the *assumptions* on that node.

The 'connector boxes' have special status. All of the other boxes in the graph can only be connected via connector boxes. The connector boxes, along with the arrows going into and out of them, can be thought of as 'communication mediums', the text in the connector box specifying precisely the attributes of that medium. Among the attributes that may apply to a medium are 'timeliness attributes' such as *timed*, *untimed*, *ordered*, *unordered*, *at-least-once*, *precisely-once* and *at-most-once*, and 'security attributes' such as *confidential*, *authenticated* and *tamperable*. When specifying the attributes of a connector box, the FT-Lib user may specify their own set of attributes, or else use pre-defined sets of attributes supplied by the library, such as 'COPPER\_WIRE' or 'WIRELESS'<sup>2</sup>.

The Dia tool records the graphical information in XML. A Dia to CSP translator maps the XML to a CSP 'signature' of a system in the form of a CSP *module*. In order to compile and check the system (against some assumption-commitment statement, c.f. Section 2.5), the module must first be *instantiated* by defining explicit *instances* of all the constituent nodes. These node instances can be modelled either in a bespoke manner or using the state-machine translators as described in Section 2.3.

The instances of constituent nodes may be 'implementations' or 'specifications'. Graphically, the difference is represented by the 'colouring' of shapes, with a (default) white-coloured shape being interpreted as an implementation, and a black-coloured shape being interpreted as the specification (of its corresponding white box).<sup>3</sup>

## 2.3 State-Machine Language

The state-machine language was conceived with the intention of enabling analysts to describe the processing of individual nodes in a system, and to do this with only a basic knowledge of state-transition machines. The eventual aim is to develop a family of translators from the state-machine language to specific modelling languages, although presently only a CSP translator is available. The individually translated processes can then be connected together using the graphical interface to form a compileable model of the complete system.

<sup>2</sup> The former is currently defined as having attributes {*timed*, *ordered*, *precisely-once*}, the latter {*untimed*, *unordered*, *at-most-once*}. These and other predefined FT-Lib mediums are, however, subject to change. Careful thought is required when defining them. One of the more exciting (and controversial?) mediums that we could conceivably be looking at in the future is single-photon lasers as used in quantum cryptography. The list of attributes may well have to be refined in order to describe adequately such 'exotic' mediums.

<sup>3</sup> The shapes that can be 'coloured' black are: rectangles (user-defined processes); tripartite rectangles (sub-systems); dashed rectangles (transient processes); stacked rectangles (N-replicated processes).

The state-machine language is a much simplified and slightly altered version of the IOA (Input-Output Automata) language [5]. State-machine filenames have the `.sm` file extension.

There are four sections to a `.sm` file; the first three of these are modelling language-neutral, and the fourth is modelling language-specific. There follows a brief overview of these sections.

The first section is the *signature* section. In this section the *inputs* and *outputs* of the state-machine are introduced by name. The *name* of the *message* being input or output is specified.

The second section is the *states* section. In this section the state variables of the machine are introduced by name and type. A state variable may be qualified by the keyword *final* - meaning that it will be constant for any particular instance of the state-machine. There are two types of reserved state variables; these are the *node\_id* and *t<n>* variables. The former is a user-defined final variable that will be used to uniquely identify the particular instances of the machine. The latter are 'clock-time' variables, *t<n>* returning the current time on clock *<n>*. If a *t<n>* variable is cited in the states section, then the machine is implicitly assumed to be a timed automaton, whether or not the variable is actually referred to in the transitions.

The third section is the *transitions* section. In this section the transitions introduced in the signature section are defined. Each transition has a *precondition* and an *effect*. The precondition is a predicate that defines when the transition is to be considered enabled. The precondition may cite state variables and, if the transition is an input transition, conditions on the inputs themselves. A precondition may be empty, meaning 'always true'. The effect defines how the state variables are to be updated as a result of the transition firing. An effect may be empty, meaning that the state variables are left unchanged.

The fourth and final section is a modelling language-specific part, in which the user may define various types, constants, 'extractor' functions, and other functions etc in particular modelling languages. These may then be referred to verbatim in the transition expressions. This section has *begin<language>* and *end<language>* delimiters, where *<language>* is the name of the modelling language, e.g. *begincsp* and *endcsp*.

Consideration is currently being given to the definition of probabilistic state-machines. This is likely to involve specifying the probability of any particular enabled transition 'firing' relative to any other enabled transition (currently the choice is determined purely by the machine's environment).

## 2.4 Templates and Core Processes

One of the original aims of the FT-Lib was to provide the means to describe and model in a re-usable fashion various 'dependability mechanisms', usually based on replication. In the current FT-Lib, these dependability mechanism 'templates' can be very conveniently described using the graphical notation - essentially graphs describing the processes and connectivity that characterise the different mechanisms. The main mechanisms under consideration by the FT-Lib to date have been N-modular redundancy, standby-systems and self-correcting mechanisms.

The provision of these templates is more than just a theoretical exercise, it means that systems that deploy the mechanisms can be modelled easily by 'importing' the relevant graphs as sub-system nodes into a Dia graph of the system (c.f. the 'sub-system' shape (tripartite rectangle) in Section 2.2). This means that all that the user is left to do is specify the instances of the 'core processes' that are cited by the templates. Sometimes, default core processes are also provided by the library, examples being a 'voter mechanism', a 'load balancer', a 'switching mechanism', etc. These default processes may be bespoke language-specific models, or they may be defined in terms of the state-machine language.

## 2.5 Specification of System Properties

It is intended that properties of systems described in the FT-Lib will be defined using the *assumption-commitment* paradigm (similar to the *rely-guarantee* paradigm of shared variable concurrency). In this approach, the specification of a node is structured into two parts: assumptions about the

behaviour of the node's environment; and commitments that should be fulfilled by the node provided that the environment fulfils these assumptions.

The FT-Lib designers consider this paradigm the most suitable for composite systems - just as an instance of a system is the 'sum of' the instances of its individual nodes, so the assumption-commitment on a system will be the 'sum of' the assumption-commitments on the individual constituent nodes.

An example of a specification using assumption-commitment might be as follows: for an OceanStore node performing the Query algorithm as described in Section 3.2.4, a node would be committed to responding to a query on the assumption that it is within a depth  $d$  of the originating request, where  $d$  is the depth of its attenuated Bloom filters, and under collision-free conditions.

## 2.6 Model Checking of Properties for Networks of Fixed Size

Once a model of a network has been constructed, it is possible to perform checks of certain properties of that network. The model-checker FDR is used to perform these checks automatically.

There are a number of forms of check that can be performed, depending on the type of property that is under consideration. The simplest form of check determines whether a system is able to perform a particular CSP event or events, which will typically indicate some erroneous behaviour of the system. The basic form of this check is as follows:

```
assert STOP [T= SYSTEM \ HideSet
```

where SYSTEM is the model of the system that we are considering, and HideSet is a set constructed so that the only events excluded from it are those that indicate the erroneous behaviour in which we are interested.

This is a traces refinement check, which asserts that the trace behaviour of SYSTEM with all events in HideSet hidden is a subset of the behaviour of the STOP process, where STOP is a special process that performs no events. The only visible events that SYSTEM \ HideSet can perform are those that indicate the erroneous behaviour occurring, so success of the refinement check indicates that that behaviour cannot occur. Conversely, if the refinement check fails, FDR will report a trace of events that lead to the erroneous behaviour occurring.

A second form of refinement check is one that determines whether a system is bound to perform some desirable behaviour, possibly given some assumptions about the environment in which the system evolves. Such a check can be phrased in the assumption-commitment style described in Section 2.5. The basic form of this kind of check is as follows:

```
assert COMMITMENT [F= (SYSTEM || ASSUMPTION) \ HideSet
```

where SYSTEM is the model of the system, ASSUMPTION and COMMITMENT are processes representing the assumption and commitment respectively, and HideSet is a set constructed so that the only events excluded from it are those that can be performed by the COMMITMENT process<sup>4</sup>.

This is a failures refinement check, which asserts two things: firstly that the behaviour of SYSTEM, when put in parallel with ASSUMPTION, and with all events in HideSet hidden, is a subset of the behaviour of COMMITMENT; and secondly that the events performed in COMMITMENT cannot be blocked from occurring. This is enough to ensure that SYSTEM will perform the desired behaviour, given the assumption (if present).

Let us take a simple example to illustrate how this kind of check can be used in practice. Consider the following very simple network graph:

---

<sup>4</sup> To formulate an assumption and commitment on a distributed system from the assumptions and commitments on its constituent nodes, we may have to suitably restrict the assumptions and commitments.



Figure 2: Simple example of a network graph

This is a network consisting of just one node connected to an environmental input and output. The node is modelled as a process called `simpleProcess`, whose state-machine definition is as follows:

```

automaton simpleProcess

signature
  inputData(m: AnyMsg)
  outputData(m: AnyMsg)

states
  time2 : ClockTimes := 0,
  t0 :    ClockTimes := 0,
  cached : AnyMsg    := NULL

transitions
  input inputData(m)
    pre cached = NULL
    eff t0 := time2;
       cached := f(m)

  output outputData(cached)
    pre cached ~= NULL;
       time2 = t0 + 1
    eff t0 := 0;
       cached := NULL
  
```

This state-machine simply receives data from the environment, stores it in its cache for one time interval (with respect to clock 2), and then outputs the data.

The state-machine and the graph can be translated into CSP, and instantiations made appropriately. We will then have a CSP process that represents the behaviour of this system, and the assumption-commitment framework can be used to check that this system behaves in the expected manner.

A check that we can perform is that some particular input, `myData` from the environment will always result in a particular output, `f(myData)` to the environment. In this case, the assumption on the behaviour of the environment would be that the particular data is transmitted to the node, and the corresponding commitment would be that the required corresponding output is made. Processes representing these two facts are as follows:

```
ASSUMPTION = SYSTEM::in!SYSTEM::inputData!myData -> STOP
```

```
COMMITMENT =
```

```
  SYSTEM::in!SYSTEM::inputData!myData ->
  SYSTEM::out!SYSTEM::outputData!f(myData) -> STOP
```

And then the check that is performed is as follows<sup>5</sup>:

```
assert COMMITMENT [F= (SYSTEM [|{|SYSTEM::in|}] ASSUMPTION) \ HideSet
```

where `HideSet` is constructed so that it contains all events except those that can be performed by `COMMITMENT` (in this case, `HideSet` will contain only clock events). FDR gives a tick in this case, indicating that the system does meet this specification.

<sup>5</sup> The parallel operator used in this assertion between `SYSTEM` and `ASSUMPTION` constrains the `inputData` events that `SYSTEM` can perform to be only those that `ASSUMPTION` can perform.

This is obviously a trivial example, but exactly the same approach can be applied to much more complicated systems and, correspondingly, much more complicated properties. In structural induction (as described in Section 4), we may want to ‘allow time’ for a partially completed network to process information from its environment before the environment inputs more data. Such an assumption on the environment would allow us to formulate simpler *Specs*.

## 2.7 Structural Induction

It is often possible to check properties of arbitrary networks through the use of *structural induction*. CSP structural induction is a *compositional*<sup>6</sup> technique for verifying certain properties of arbitrary large systems by verifying only a finite number of *base* and *step* cases. The following description is a digest of the description to be found in [4].

For structural induction to be applicable, we must be able to reason that any single usage of the service is a *network invariant* - meaning that the specification of the property is the same for all (sufficiently large) networks.

Suppose we want to show that a property *Spec* (the network invariant) holds of some network no matter how large the network. Let  $Sys_i, i \in I$  be a finite set of network models and suppose that the following hold for some CSP operator  $\circ$  and all  $i \in I$ :

$$Spec \sqsubseteq Sys_i \tag{1}$$

$$Spec \sqsubseteq Spec \circ Sys_i \tag{2}$$

then it would follow that:

$$Spec \sqsubseteq Sys_{i_1} \circ Sys_{i_2} \circ \dots \circ Sys_{i_n} \tag{3}$$

for any  $i_j$  from  $I$ <sup>7</sup>. I.e., the property *Spec* holds for any system composed from an arbitrary number of the systems  $Sys_i$  using the operator  $\circ$ . Usually  $\circ$  is shared parallel,  $\parallel$ , interleaving,  $\parallel\parallel$ , or piping, [*left*  $\leftrightarrow$  *right*].

In Section 4 we will be discussing what support we might provide for structural induction in the future FT library framework. Possible support will be along the lines of a library of ‘basic building blocks’ representing the  $Sys_i$  in the structural induction arguments.

---

<sup>6</sup> “‘Composition’ in the strict CSP sense of composing implementation and specification processes from a number of simpler sub-processes usually using the shared-parallel or interleaving operators.” [4]

<sup>7</sup> (1) are called the *base cases*, (2) are called the *step cases*.

## 3 A Case Study: OceanStore

### 3.1 Overview of OceanStore

OceanStore is an architecture for a global-scale, persistent, distributed storage mechanism that was designed by a team at the Computer Science department at the University of California, Berkeley. Many components of OceanStore have been implemented and are already functioning in isolation; a complete prototype is currently being developed.

OceanStore was designed with two goals in mind. The first of these is that it can be constructed from an entirely untrusted infrastructure, and that it is therefore resilient to server crashes and information leakage to third parties. In order that such an infrastructure can be used without data being compromised, all data is protected through redundancy and cryptographic techniques.

The second goal is that OceanStore supports *nomadic data*. Nomadic data is defined to be data that is allowed to flow freely. In a system as large as OceanStore, locality of data is of extreme importance; therefore, a stated aim is that data may be cached *anywhere, anytime*.

The elements of OceanStore that we have modelled using the fault-tolerance library are related to data location and routing. The mechanism that performs these processes is fairly sophisticated, as a result of the fact that objects in the OceanStore are free to reside at *any* of the servers. This fact allows flexibility in selecting policies for replication, availability, caching, etc., but has the side-effect that the process of locating these objects is rather complicated.

The mechanism used is a two-tiered approach, with a fast probabilistic algorithm [7], backed up by a slower, reliable hierarchical method. The reason for this is that objects that are accessed frequently are likely to be located near to where they are being used. The probabilistic algorithm may route to objects rapidly if they reside nearby, but success is not guaranteed. So if this attempt fails, a large-scale hierarchical data structure locates objects wherever they are in the OceanStore.

The focus of our attention so far has been the probabilistic data location and routing algorithm; this is described in the next section.

### 3.2 Probabilistic Data Location and Routing

In order to perform the probabilistic data location and routing algorithm, each server must maintain a set of neighbours. A server associates with each neighbour a probability of finding each object in the system through that neighbour. This association is maintained efficiently (and in constant space) using an *attenuated Bloom filter*, a data structure based on a *Bloom filter* [3]. The set of these probabilities forms a potential function over the servers in the network; to locate an object, this function is climbed through the network until the object is found.

#### 3.2.1 Bloom Filters

Bloom filters are an efficient, lossy way of describing sets. A Bloom filter is a vector of bits, of width  $w$ . It is associated with a number of independent hash functions, each of which maps to the range  $[0, w-1]$ . To represent a set of elements as a Bloom filter, each element is hashed, and the bits of the vector that correspond to the results are set.

To determine whether a set represented by a Bloom filter contains a particular element, that element is hashed and the corresponding bits in the filter are examined. If any of those bits in the filter are not set, the set definitely does not contain the element; false negatives are therefore impossible. However, if all of the bits are set, the set *may* contain the element; but there is a non-zero probability that it does not. This case is called a false positive. Figure 3 shows a sample Bloom filter.

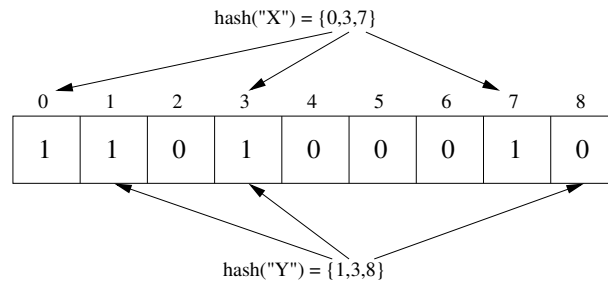


Figure 3: A Bloom Filter. An array of  $w$  bits that summarise a set of objects. To check whether an object is present in the set, the object's name is hashed with  $n$  different hash functions (here,  $n = 3$ ), and the bits corresponding to the result are checked in the Bloom filter. In this example, the set probably contains "X", because bits 0, 3 and 7 are all true. However, it definitely does not contain "Y", because bit 8 is false.

If the cardinality of the represented set is a significant fraction of the width, the Bloom filter becomes *overloaded*. If this is the case, the rate of false positives is so high that the filter is essentially useless. One solution is to use a wider filter, but this has the effect of increasing the amount of storage required. Thus there is always a trade-off between accuracy and storage.

### 3.2.2 Attenuated Bloom Filters

An attenuated Bloom filter of depth  $d$  is an array of  $d$  normal Bloom filters. In the probabilistic algorithm, each neighbour link is associated with an attenuated Bloom filter. The first filter in the array summarises objects stored at that neighbour, i.e. it is precisely the Bloom filter that represents those objects. The  $i$ th filter in the array is the merging of all Bloom filters for all of the nodes  $i$  hops along any path starting at that neighbour link. Figure 4 represents a network of four nodes, and shows the attenuated Bloom filter that Node  $A$  would associate with Node  $B$  in this network. For example, both "W" and "Z" are two hops away from Node  $A$  through Node  $B$ , so the second level of filter  $F$  contains true values at all bits in the union of those objects' hash values (0, 2, 3, 5, 7).

To determine the potential value for an object, one examines each level of the attenuated Bloom filter for the hashes of the object's name. The value of the potential function for a given object is the sum of all the potential values for the levels of the filter that contain the hash values. For example, in  $F$ , the object "W" would map to the potential value  $1/4 + 1/8 = 3/8$ .

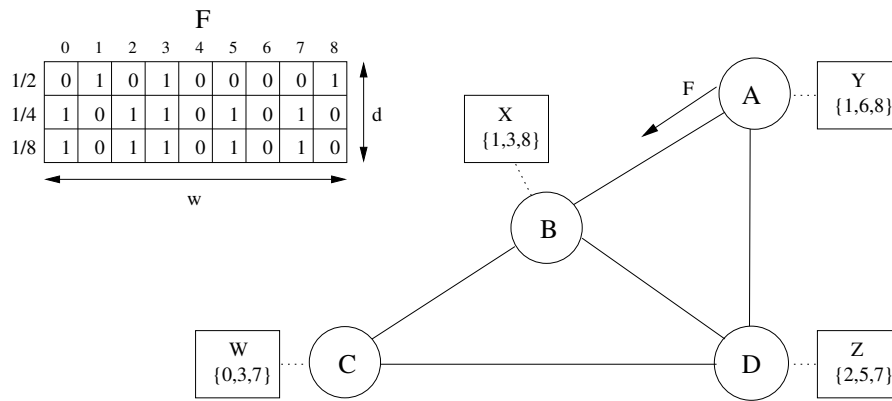


Figure 4: Attenuated Bloom Filters. An attenuated Bloom filter is an array of  $d$  Bloom filters, each of width  $w$ . Each outgoing link (for example,  $A \rightarrow B$ ) has an attenuated filter associated with it ( $F$ , in this case). The first level of the filter summarises objects present at the neighbour at the end of the link, and the second level summarises objects that are two hops away along that link, etc. A potential value is assigned to each level (here  $\frac{1}{2}$ ,  $\frac{1}{4}$ , etc.).

### 3.2.3 The Update Algorithm

For servers to have a chance of locating data stored in the local network, the attenuated Bloom filters stored at each node must be kept up-to-date. Each time a new piece of data is added to a server, it is possible that the Bloom filter representing the set of data items it stores will change as well. This change must be propagated to them in some manner; the method used is the *update algorithm*.

The most important observation to be made about the update algorithm is that an update to a server with attenuated Bloom filters of depth  $d$  should eventually result in a change to at least one bit in the filters of every server within  $d$  hops of the original server. This will occur as long as the Bloom filters are not loaded to such a degree that they are no longer useful for location. Ideally, this propagation is a single wave spreading from the source of the change outward.

An update proceeds as follows. Every server in the system stores both an attenuated Bloom filter for each outgoing link, and a copy of its neighbour's view of the reverse direction. When a new piece of data is stored, the server calculates the changed bits in its own filter and in each filter its neighbours maintain of it. It then sends these bits out to each neighbour. On receiving such a message, each neighbour attenuates the bits one level and calculates the changes they will make in each of its own neighbours' filters. These changes are then sent out as well. The update continues to be propagated until the last level of a receiving node's attenuated Bloom filters is reached.

### 3.2.4 The Query Algorithm

To perform a location query, the querying node examines the first level of the attenuated Bloom filter associated with each of its neighbour links. If any of the filters matches, it is likely that the desired data item is located at one of the corresponding neighbours, and the query is forwarded to the nearest one. If no match is found, the querying node examines the second level of each attenuated Bloom filter, and if there is a match, forwards the query to the nearest matching neighbour. In this case, it is not the immediate neighbour who is likely to possess the data, but one of its neighbours, and so the algorithm proceeds as before, with the current server examining its own stored attenuated Bloom filters.

A filter of depth  $d$  by definition stores information only about servers  $d$  hops from the current server. For this reason, there is no incentive to propagate a query for more than  $d$  hops. When such circumstances arise, the normal action is to give up and defer to the deterministic algorithm.

### 3.3 Modelling

In order to keep the state space of our model of the probabilistic algorithm to manageable levels, we were forced to split it into two submodels, the first being a model of the update algorithm, and the second a model of the query algorithm.

#### 3.3.1 Model of Update Algorithm

The full code for the state-machine specification of a node performing the update algorithm is given in Appendix A.1. However, the graph for a fully connected network of three nodes performing this algorithm is shown in Figure 5. This graph shows the nodes connected by bidirectional links, with each node having a user input and output. One of the nodes also has an environmental input and output, which allows communication with other networks. The names of the input and output transitions of each node are also shown.

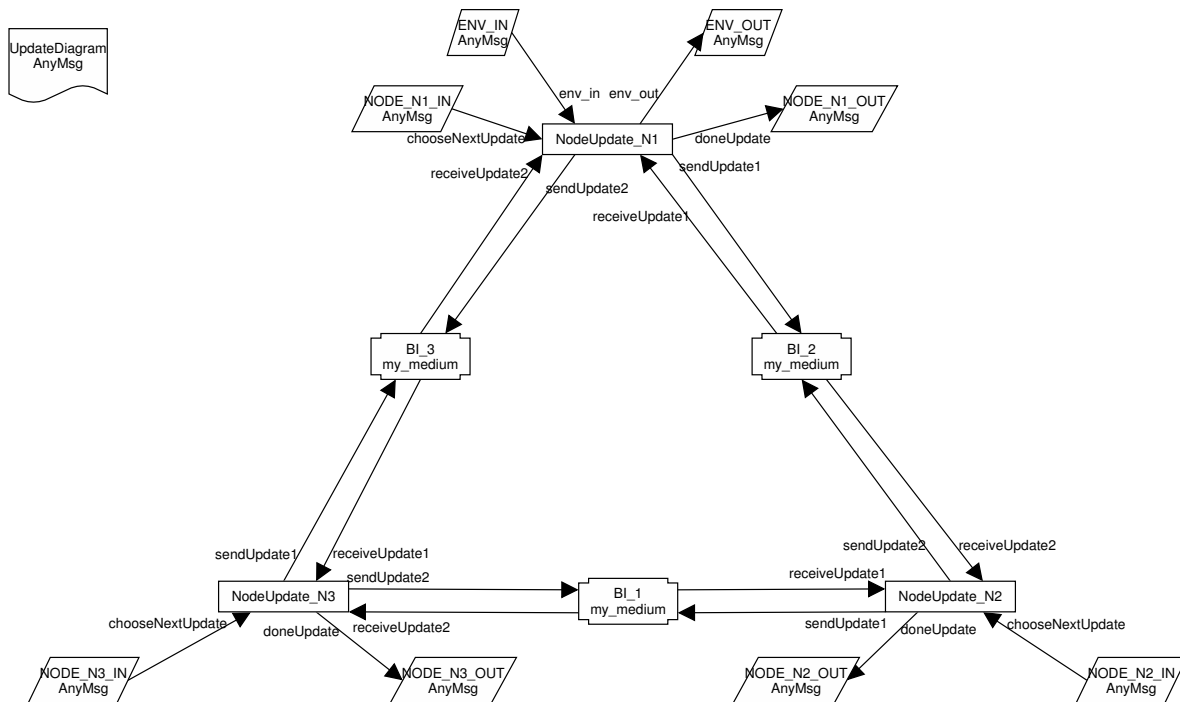


Figure 5: Network graph for update algorithm

In our model, each node has two hash functions, and an initial set of data; these will be defined after the state-machine specification has been translated into CSP, when instantiations are made of the resultant process. Each node also has a set of neighbour/attenuated Bloom filter pairs, which are initially empty. The nodes also maintain a number of other variables, which contain such values as a set of updates that need to be sent out, the name of the update the node has chosen to send out next, and whether or not the node has completed the algorithm.

The first event defined in our state-machine specification is *chooseNextUpdate*, which is an input transition that is available to all nodes. This transition has the effect of a node deciding which update to send out next, and storing the name of that update.

The second input event is *receiveUpdate*. This transition specifies the behaviour of a node on receiving an update from a neighbour. It has three effects, the first of which is that the node recalculates appropriately the attenuated Bloom filter that is associated with the link to the sender of the update, to take account of the update. The second effect deals with the forwarding (or otherwise) of the update. If the update has already been forwarded as many times as the depth

of the attenuated Bloom filters, or if the node has no neighbours other than the one that sent the update, the update is not forwarded. Otherwise, a new update is added to that node's set of updates to be sent out, for each neighbour of the node other than the one that sent the update. The third effect determines whether or not the node is able to announce that it has completed the algorithm; it is only able to announce completion if it has no more updates to send out.

The first of the output events is `sendUpdate`. Here the behaviour of a node on sending an update is specified. The precondition for this transition to occur is that the node has chosen its next update to be sent out (by performing a `chooseNextUpdate` transition). After the event has occurred, the update is removed from the set of updates to be sent out, and the name of the next update to be sent out is reset to null. If the node has no more updates to send out, the node is then ready to announce that it has completed the update algorithm.

The final event is `doneUpdate`. This event represents a node announcing that it has completed the algorithm. The node is able to perform this event only if it has no more updates to send out.

Following translation of the state-machine description into CSP, three instantiations of it are made, one corresponding to each node in the network. It is in making the instantiations that the hash functions and the names of data present at each node are specified. A function is also defined that associates state-machine transitions with labels on the edges of the network graph.

### 3.3.2 Model of Query Algorithm

We now describe our model of a node behaving according to the query algorithm (again the full state-machine specification code is given in Appendix A.2). The network graph for three fully connected nodes performing the query algorithm is shown in Figure 6. This graph is similar to that for the update algorithm, the only difference being the transition names. This similarity is to be expected, as in reality the nodes would perform both algorithms concurrently over the same network.

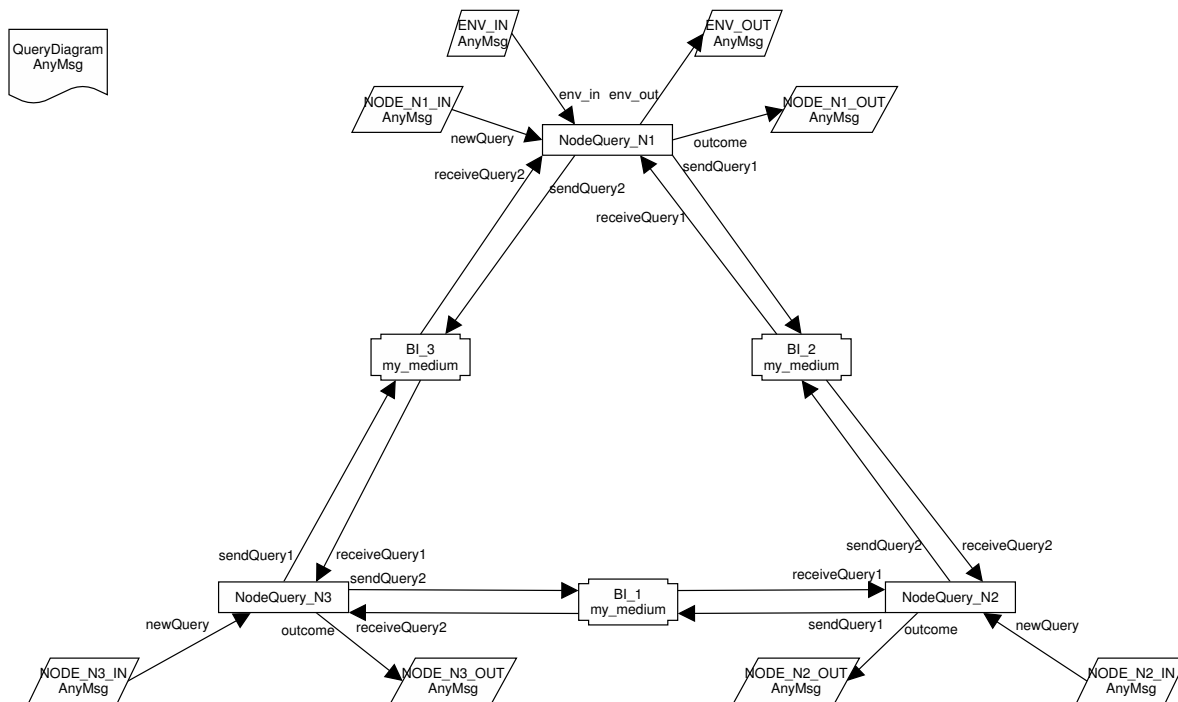


Figure 6: Network graph for query algorithm

As in the update algorithm, each node has two hash functions and a set of data. In addition, each node has a set of constant neighbour/attenuated Bloom filter pairs, which may be initialised to be

correct or incorrect, depending upon which properties of the algorithm we wish to check. The nodes also maintain a query buffer, which is initially empty; in addition, a limit on the size of the buffer must be specified (this is to reduce the state space of the resultant process). Nodes also maintain a variable that indicates the result of the most recently completed query at that node.

The first input event in our specification is `newQuery`, which specifies the behaviour of a node initiating a query. This event is only enabled if the query buffer is not already full, while a node may initiate a query for any data that it does not hold itself. The effects of this event determine whether a query should be added to the query buffer, and whether or not failure of the query should be announced immediately. If there is no attenuated Bloom filter that contains each of the hashes of the data, the query has failed immediately, so the query buffer is left unchanged, and the node is then able to announce failure of its query. If any attenuated Bloom filters do contain the hashes, a query is added to the query buffer, to be sent to the node with the smallest ID (this arbitrary choice of node is made because there is no concept of the 'nearest' node in our model).

The second event is `receiveQuery`, which describes the behaviour of a node receiving a query from another node. There are three possibilities for the effect of this transition. Firstly, if the data being searched for is present at this node, the query buffer is left unchanged and the node is left able to announce that the query was concluded successfully; secondly, if the data is not present at this node but the stored attenuated Bloom filters indicate that it may be present at routes through one or more neighbours, a query is added to the query buffer with its destination being the node with the smallest node ID of those that matched the hashes in the query, and the variable that stores the result of the most recent query is made null; finally, if the data is not present at this node, and either this node's query buffer is full or no attenuated Bloom filters match the hashes in the query, the query buffer is left unchanged and the result variable is made false.

The first of the output events is `sendQuery`. Its effect is simply to send out the head of the (non-empty) query buffer.

The final event is `outcome`, which outputs the result of the most recent query to be completed at that node.

As in the update algorithm, the state-machine description is translated into CSP, and an instantiation of the resultant process is made for each node in the network. Values are given to the constants stored at each node, and the state-machine transitions are associated with network graph labels.

## **3.4 Results for Networks of Fixed Size**

### **3.4.1 Update Algorithm**

The check we have performed on our model is that the update algorithm will always terminate, i.e. the events that indicate that each node has completed the algorithm will always be performed for the given distribution of data. We phrase this check in the assumption/commitment style. The commitment is that the update algorithm will always terminate, and no assumption needs to be present.

```
NodeMappings =
  {NODES_UPDATE1::box_mapping(n) | n <- {NODES_UPDATE1::NodeUpdate_N1,
    NODES_UPDATE1::NodeUpdate_N2,NODES_UPDATE1::NodeUpdate_N3}}

COMMITMENT =
let
  COMMITMENT_(mappings) = NODES_UPDATE1::out?m:mappings!
    NODES_UPDATE1::doneUpdate!R!true -> COMMITMENT_(diff(mappings,{m}))
within
  COMMITMENT_(NodeMappings)

HideSet =
```

```
diff({|p_in,p_out,NODES_UPDATE1::in,NODES_UPDATE1::out|},
     {|NODES_UPDATE1::out.n.NODES_UPDATE1::doneUpdate |
      n <- NODES_UPDATE1::box_nos|})
```

```
assert COMMITMENT [F= NODES_UPDATE \ HideSet
```

This check was performed on a number of different versions of the model, including all possible distinct arrangements of three pieces of data over a three node network (by symmetry, there are only a small number of distinct arrangements). The check passed for each version of the model on which it was performed, indicating that the update algorithm is guaranteed to terminate for this network topology.

### 3.4.2 Query Algorithm Results

The check we have performed on this model is whether or not any events that indicate failure of a query can occur.

```
HideSet =
  diff({|p_in,p_out,NODES_QUERY1::in,NODES_QUERY1::out|},
       {|NODES_QUERY1::out.n.NODES_QUERY1::outcome.R.m.d.false |
        n <- NODES_QUERY1::box_nos, m <- MyNodes, d <- Data|})
```

```
assert STOP [T= NODES_QUERY \ HideSet
```

This check was performed on several different versions of the model, obtaining a number of different results. It was found that if the attenuated Bloom filters were correct and not overloaded to the extent that false positives were obtained, and the data being searched for was present in the network, the query algorithm was guaranteed to work for this network topology.

However, because of its very nature as a probabilistic algorithm, it is known that the query algorithm is not guaranteed to work in all cases. After changing our model so that either the attenuated Bloom filters were overloaded or they were initialised incorrectly, the failure of queries was observed.

## 4 Structural Induction in CSP: Possibilities for FT Library Support and Application to OceanStore

### 4.1 Rationale

OceanStore, like several other protocols of interest to FORWARD, such as Grid [2], is an ad-hoc protocol that is designed to run on arbitrary large networks. Ideally, we would like to formally verify such protocols for arbitrary large networks, and *structural induction* is one means by which we might achieve this this, at least for certain classes of property.

A large number of authors have demonstrated that induction is a method which can be successfully used in the analysis of distributed systems. Structural inductions are all variants on the following basic inductive technique; to show that property  $P$  (the network invariant) holds of some network no matter how large the network:

- Show firstly that the network of size 1 satisfies property  $P$ . This is the base case.

$$\text{network size} = 1 \Rightarrow P$$

- Secondly, show that if  $P$  is true of the network with size  $k$ , then  $P$  is true of the network with size  $k + 1$ . This is the step case.

$$\forall k \bullet P[k / \text{network size}] \Rightarrow P[k+1 / \text{network size}]$$

The OceanStore Update and Query models reported on so far in this document involve a relatively small three node loop topology. One way in which we might scale the models would be to verify the algorithms for various other topologies, chosen so as to afford the best possible coverage. Then, by a 'pen and paper' argument we might look to extrapolate the results to classes of arbitrary large networks.

CSP structural induction provides a more formalised alternative that obviates the need for a pen and paper proof<sup>8</sup>. However, CSP structural induction proofs require some degree of expertise to formulate correctly. In this section, we shall look at ways in which the FT Library may afford support to structural induction proofs, to make them easier to formulate.

### 4.2 Support for CSP Structural Induction of OceanStore

In this Section we shall discuss two of the three main issues involved in a structural induction proof, namely deriving  $Sys_i$  suitable for base and step case statements, and the correct formulation of those statements from a given set of  $Sys_i$  and a  $Spec$ <sup>9</sup>. Throughout, we shall be discussing what support is currently afforded by the library, and shall propose future support. The discussion is best made with frequent references to an example, and for this we shall use the OceanStore Update and Query algorithms - bearing in mind that we would want any future work to be sufficient, at least, to support structural induction on those algorithms.

#### 4.2.1 Formulating Suitable $Sys_i$

The  $Sys_i$  of Equations 1 and 2 in Section 2.7 are inextricably linked to the invariant property, represented by  $Spec$ . Ideally, the  $Sys_i$  will be minimal in the sense that the property represented by  $Spec$  holds of a  $Sys_i$  (Equation 1), although it does distinguish between the nodes of the  $Sys_i$ , but does not hold of a sub-topology of the  $Sys_i$ .

What support might be afforded the analyst in the construction of  $Sys_i$  applicable to a structural induction proof? The onus, almost certainly, will be on the analyst to proffer candidate minimal

---

<sup>8</sup> Pen and paper proofs may be difficult to formulate convincingly.

<sup>9</sup> The third issue, the formulation of a suitable  $Spec$ , is discussed in 4.3.

topologies, but we could reasonably provide support in the FT Library by way of a library of graphical templates representing the more commonly used of those minimal topologies.

Let us consider how this might work for the OceanStore Update algorithm. Here, our goal might be to show that for arbitrary large networks, any particular run of the Update algorithm following a user's *chooseUpdate* completes properly in the sense that precisely the right number of nodes declare a *doneUpdate*.<sup>10</sup>

In the above, the 'right number' that is alluded to will in general be dependent on the depth of the Bloom filter and characteristics of the network graph. To see this, let us suppose the Bloom filter has depth  $d$ . Then, the number of nodes involved in an Update request will be the number of nodes within  $d$  hops of the node that received the user *chooseUpdate* request. In order for a structural induction proof to work, we need to ensure that this number is a network invariant, and the only way to do this is to impose a bound on the number of neighbours that each node in the network has. That is, we must make the assumption that there is some number  $K$ ,  $K > 0$ , such that for all nodes  $N$  in the network, the number of neighbours of  $N$ , i.e. the *degree* of  $N$ , is no more than  $K$ . Then, it is not hard to prove that the number of nodes involved in the processing of any single *chooseUpdate* will be no more than  $1 + K \times (\sum_{0 \leq i \leq (d-1)} (K-1)^i)$ . Here, the maximal case occurs when the subgraph of radius  $d$  centred at the node receiving an Update request from its user is a tree with each node having degree  $K$ . Graphs in which all nodes have precisely degree  $K$  are called *K-regular* graphs. Of course, for simpler networks the number of nodes involved will be much lower.

The only 2-regular connected networks are the very important class of loop-topologies, which are pertinent to a natural first generalisation of our existing three node loop OceanStore network to loops of arbitrary lengths. Here, according to the above calculation, the minimal *Sys<sub>i</sub>* should be subgraphs of loops involving up to  $1 + 2d$  nodes. However, nodes only have to distinguish between themselves and between their two neighbours in order to process an Update correctly. For algorithms with this characteristic, *Sys<sub>i</sub>* of up to 3 nodes would seem to be sufficient, and to cover the case of loops involving an arbitrary multiple of three nodes, at most two are required.

Below, as an example, we show how two such *Sys<sub>i</sub>* - *Sys<sub>1</sub>* and *Sys<sub>2</sub>*, say - might be depicted graphically. Informally, *Sys<sub>1</sub>* will be used to extend on the right two parallel chains by adding a new 'link', while *Sys<sub>2</sub>* will be used to 'glue' the two ends of these chains together to form a loop.<sup>11</sup>

Figure 7 illustrates a number of *Sys<sub>1</sub>*s being composed (in CSP, the compositional operator that we envisage using will be piping). The square boxes, in accordance with the FT Library syntax, indicate the separate nodes running an algorithm,  $P$ , with the characteristic that it only needs to distinguish between the host node and between its host's neighbours. Each node has been carefully allocated an 'identifier' taken from  $\{1, \dots, 3\}$  sufficient for it to make these distinctions. The dotted lines indicate the interfaces between the *Sys<sub>1</sub>*s. Arrows crossing the dotted lines indicate the 'traffic' between the *Sys<sub>1</sub>*s, i.e. the data and the direction of the data sent into and out from each *Sys<sub>1</sub>*. Here, the arrows labelled  $c_{i-j}$  may be thought of as one-way communications channels conveying messages from node  $i$  to node  $j$ .

<sup>10</sup> The properties that one typically wishes to verify using structural induction are often misleadingly 'obvious'. For the OceanStore Update algorithm, the question of whether it is even deadlock free or livelock free in a loop topology is not obvious. To ensure the latter property, for example, would require guards in the nodes' processing to the effect that Updates are not forwarded to some neighbours, and that an Update request is only processed if the data item has not already been seen. If, in addition, we start to introduce faulty behaviour to test the fault-tolerance of the system - as discussed in 4.3.3 - then matters will become even more complicated.

<sup>11</sup> NB. There is an arguably simpler way of linking together individual nodes to form a loop, namely by forming a chain of individual nodes, then gluing together the two ends. The *Sys<sub>1</sub>* and *Sys<sub>2</sub>* described here are experimental. It may be that they fit in better with a more general framework, and are more applicable in the investigation of fault tolerance.

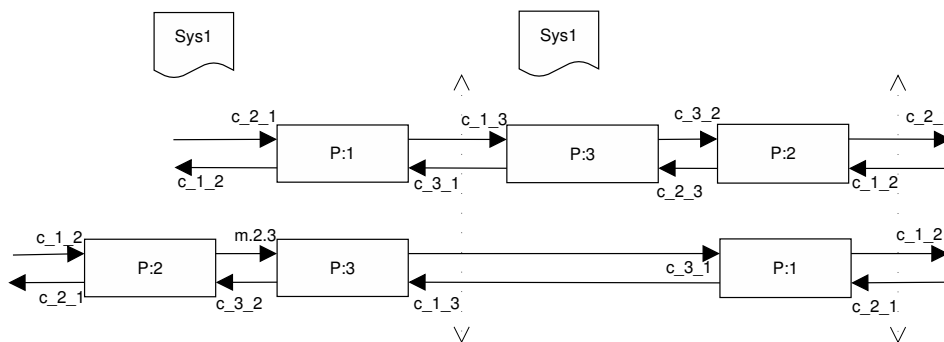


Figure 7: Using a  $Sys_1$  to add another link onto the right side of two parallel chains

Figure 8 illustrates a  $Sys_1$  composed with (i.e. piped to) a  $Sys_2$ , thereby ‘closing’ one end of the loop.

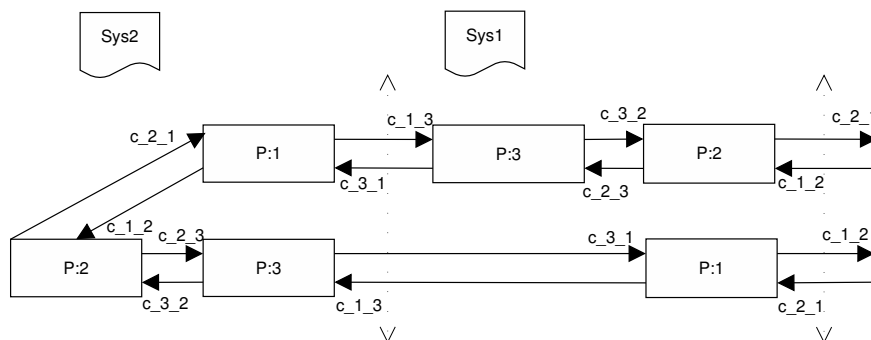


Figure 8: Using  $Sys_2$  to close the left ends of two parallel chains

As mentioned above, care must be taken allocating the node identifiers, even for this relatively simple loop topology. Suppose, in general, that the process being verified is such that each node running the process needs to distinguish between  $N_d$  nodes including itself. Then the derivation of the  $Sys_i$  (including the allocation of appropriate identifiers and channels) will become increasingly complex as  $N_d$  increases.

The provision of a set library of  $Sys_i$ s (with identifiers already allocated and arrows correspondingly labelled) would, therefore, be of considerable assistance in structural induction proofs. We emphasise that this proposed library of graphical representations of the  $Sys_i$ s would say nothing about the processes being deployed - which are simply represented in the graphs by a tag ( $P$  in the case of  $Sys_1$  and  $Sys_2$ ). Nor would they say anything about the semantics of the messages being communicated between nodes. Rather, they would be truly generic templates, describing the basic communications topology ‘building blocks’ sufficient to build different arbitrary large networks which, by construction, would be amenable to structural induction proofs. We conjecture that these  $Sys_i$  will be characterised by: (i) the number of nodes in the  $Sys_i$  that interface with the environment<sup>12</sup>; (ii) the maximum number of nodes that any particular node in the network is expected to have to distinguish between, including itself<sup>13</sup>.

<sup>12</sup> Pictorially represented by the number of arrows crossing the dotted line.

<sup>13</sup> This is the cardinality,  $\#nodes$ , of the node identifier set. It may be that only some  $Sys_i$  can support the distinction of  $\#nodes$ .

## 4.2.2 Formulating the Base and Step Cases

For the  $3N$  loop-topologies constructed using  $Sys_1$  and  $Sys_2$  of 4.2.1, there is one base case and two step cases as follows.

The base case is:

$$Spec \sqsubseteq Sys_2 \quad (4)$$

and the step cases are:

$$Spec \sqsubseteq Sys_2[left \leftrightarrow right]Spec \quad (5)$$

and

$$Spec \sqsubseteq Sys_1[left \leftrightarrow right]Spec \quad (6)$$

In the above, *left* is a channel coming from a  $Sys_i$ 's environment to the  $Sys_i$ , while *right* is a channel from a  $Sys_i$  to its environment. With reference to figures 7 and 8 the left channel will be a single channel multiplexed to the channels crossing into the  $Sys_i$  through a dotted line. The right channel is a single channel de-multiplexed from the channels going out from the  $Sys_i$  through a dotted line.

As we can see from the above, the exact nature of the base and step statements will depend on the topologies that we are interested in. The single base case and two step cases stated above are sufficient for loops, but other combinations could give rise to quite different topologies (still, of course formed from the  $Sys_i$ ), very possibly not even connected.

Suppose we have an invariant property represented by a given *Spec* process plus a given set of  $Sys_i$ . Suppose we had a convention - possibly graphical - for representing the different classes of arbitrary large topologies that we wanted to form from the building block  $Sys_i$ . Then the corresponding refinement assertions could be automatically generated from the pictorial representation, the  $Sys_i$  and the *Spec* (and then, possibly, run in FDR's batch mode). At present, the FT Library can capture graphically, and in a natural way, the *Spec* and  $Sys_i$  combinations appearing on the right hand side of the step cases in the case when the operator is piping.

## 4.3 Other Future Work

In Section 4.2 we discussed the formulation of suitable  $Sys_i$  and the formulation of the base and step statements. That discussion was tailored towards the OceanStore example, though we indicated where the theory was suitable for more general networks and algorithms. In this section we shall look at further generalisations beyond what we would need for OceanStore.

### 4.3.1 Non-Symmetrical Networks

For the  $Sys_i$  library discussed above, we envisage providing  $Sys_i$  sufficient to construct the more commonly seen networks built-up from loop, star, chain and tree topologies. In this section we shall briefly discuss the pros and cons of structural induction of more 'exotic' topologies.

The  $K$ -regular connected graphs when  $K = 2$  are simple loop topologies. However, the connected  $K$ -regular graphs become increasingly complex as  $K$  is increased. It is unclear whether much benefit would be gleaned from providing support for the inductive construction of arbitrary connected  $K$ -regular networks if the graphs are so symmetric and complex as to be highly unlikely to occur in practise.

The difference in the structural induction proofs for the different  $K$ -regular systems is likely to be manifest 'only' in the number of basic  $Sys_i$ s - analogous to the  $Sys_1$  and  $Sys_2$  of 4.2.1 - that are needed to construct the graphs (up to isomorphism). Although that may be acceptable in theory, the basic  $Sys_i$  will become increasingly complex and numerous as  $K$  increases<sup>14</sup>.

The connected subgraphs of  $K$ -regular graphs is the most general class of graphs that one could conceivably be interested in, consisting of all connected graphs with the property that each node has degree at most  $K$ .

---

<sup>14</sup> Possibly increasing exponentially with  $K$ .

Here matters are complicated by the fact that we are likely to get an increasing number of 'special case' nodes in the  $Sys_i$  whose behaviours have to be accounted for separately when writing the  $Spec$ . With reference to  $Sys_2$  of Section 4.2.1, a likely 'special node' - if any - would be the node with identifier 2. That node is positioned between nodes 1 and 3 of  $Sys_2$  and does not interface with the environment at all. It may be that the visible actions of such 'special' nodes have to be flagged in some way in order for a structural induction  $Spec$  process to account for its difference in behaviour relative to the other nodes in the  $Sys_i$ . There are certainly many ways in which that can be achieved in the FT Library, but the  $Spec$  will become unavoidably more complex the more 'special cases' it has to account for.

In summary, it is likely that considerable work would be needed to account for the most arbitrary  $Sys_i$ . It is arguable that the effort would be better spent providing a sound framework for the structural induction of the more common networks cited at the beginning of this section.

#### 4.3.2 Formulating a suitable $Spec$

The derivation of a  $Spec$  suitable for a structural induction proof requires a great deal of care. The  $Spec$  appears on both sides of the step cases and must specify what every node must do on an input from inside its  $Sys_i$  and from outside the system - in short, there may be many different cases to account for. The FT Library's state-machine language and CSP translator may help in this process. We can speculate that the  $Spec$  might comprise of a number of state machines in parallel - one for each node that might appear in the  $Sys_i$ . These machines would effectively be the specifications for the node processes, possibly under some *assumption* of behaviour from the rest of the system (the rest of the  $Sys_i$  and the  $Sys_i$ 's environment) and guaranteeing some *commitment*. The use of assumptions may turn out to be of crucial importance in simplifying the spec process.

#### 4.3.3 Investigating Fault-Tolerance

It is a simple matter to capture graphically within the FT Library a node's propensity to fail in one of a number of 'standard' ways, including stop-failure and Byzantine failure.

We can envisage making certain nodes within our  $Sys_i$  building blocks failure-prone in order to investigate the relative fault-tolerance of different arbitrary large topologies built from different  $Sys_i$ . Here, the OceanStore Query algorithm would make for an interesting case study. For that algorithm, one might ask, for example: are the systems formed from a set of  $Sys_i$  'fault tolerant' in that an OceanStore query always succeeds, even if a particular number of links in each  $Sys_i$  were to stop-fail concurrently?<sup>15</sup>. If yes, then how many links need to fail in the  $Sys_i$  before some Query does fail?

#### 4.3.4 A Graphical Representation for Inductively Constructed Systems

We would like a simple graphical representation for arbitrary large systems built by a structural composition argument. This graphical representation - a shape of some sort - could then be incorporated in a network graph of a larger system. Whatever the shape that is chosen, some text in the shape should unambiguously cite the  $Sys_i$  involved, and the class of networks that are being constructed from those  $Sys_i$ .

The proposed shapes will represent an arbitrary large inductively constructed system. However, in any automated CSP proof citing that system, only one (necessarily finite) instance of the system can be used. So the question arises as to what instance(s) of the system - let us call it  $Sys_{Inductive}$  - should we use. To answer this, we note that the only properties that the rest of the system should be depending on the  $Sys_{Inductive}$  to provide will be invariants of the different  $Sys_{Inductive}$ . That is simply because we could, anyway, only reason about invariant properties of the class of  $Sys_{Inductive}$

---

<sup>15</sup> Presumably the query will succeed provided the query can propagate through the co-joining  $Sys_i$  in such a way as to bypass any failed node

systems. Hence, any instance of the  $Sys_{Inductive}$  should suffice in our automated proof, presumably even the  $Spec$ .

## 5 Conclusions

This report has introduced a framework for the modelling and analysis of distributed systems, and illustrated the use of this framework through the analysis of OceanStore's probabilistic data location and routing algorithms.

Some simple correctness properties of these algorithms have been formally verified using FDR, for a particular network of finite size. In particular, it has been shown that the probabilistic update algorithm is guaranteed to terminate successfully for this network; it has also been shown that queries propagated according to the probabilistic query algorithm are, under certain specified conditions, guaranteed to be resolved successfully.

We have discussed in detail the potential for scaling the OceanStore models to various arbitrary large networks using CSP structural induction. We looked at ways in which the FT Library could provide semi-automated support for the formulation of structural induction arguments.

# A OceanStore State Machine Scripts

## A.1 Update Algorithm

automaton NODE\_UPDATE

```
signature
  input  chooseNextUpdate(m: {|U|})
  input  receiveUpdate1(m: {|U|})
  input  receiveUpdate2(m: {|U|})
  output sendUpdate1(m: {|U|})
  output sendUpdate2(m: {|U|})
  output doneUpdate(m: {|R|})

states
  final data : Set(Data),
  final hash1 : HashFn,
  final hash2 : HashFn,
  link_att_filters : Set((MyNodes,Seq(Seq(Bool)))) :=
    make_att_filters(my_nbrs(node_id,Links)),
  to_be_sent : Set(AnyMsg) := make_updates(node_id,hash1,hash2,data,my_nbrs(node_id,Links)),
  next_update : AnyMsg := NULL,
  done : AnyMsg := if empty(make_updates(node_id,hash1,hash2,data,my_nbrs(node_id,Links)))
    then R.true
    else R.false

transitions
  input chooseNextUpdate(m)
    pre next_update=NULL;
      member(m,to_be_sent)
    eff next_update := m

  input receiveUpdate1(m)
    pre member(get_sender(m),my_nbrs(node_id,Links));
      get_receiver(m)=node_id;
      member(get_hashes(m),possible_pairs_of_hashes(hash1,hash2))
    eff link_att_filters :=
      add_to_link_att_filters2(<get_hash1(m),get_hash2(m)>,get_level(m),
        get_sender(m),link_att_filters);
    to_be_sent :=
      if get_level(m)==att_filter_depth or
        empty(diff(my_nbrs(node_id,Links),{get_sender(m)}))
      then to_be_sent
      else union(to_be_sent,{U.node_id.nbr_node_id.get_hashes(m).(get_level(m)+1)|
        nbr_node_id<-diff(my_nbrs(node_id,Links),{get_sender(m)})});
    done :=
      if empty(to_be_sent) and (get_level(m)==att_filter_depth or
        empty(diff(my_nbrs(node_id,Links),{get_sender(m)})))
      then R.true
      else R.false

  input receiveUpdate2(m)
    pre member(get_sender(m),my_nbrs(node_id,Links));
      get_receiver(m)=node_id;
      member(get_hashes(m),possible_pairs_of_hashes(hash1,hash2))
    eff link_att_filters :=
      add_to_link_att_filters2(<get_hash1(m),get_hash2(m)>,
        get_level(m),get_sender(m),link_att_filters);
    to_be_sent :=
      if get_level(m)==att_filter_depth or
        empty(diff(my_nbrs(node_id,Links),{get_sender(m)}))
      then to_be_sent
      else union(to_be_sent,{U.node_id.nbr_node_id.get_hashes(m).(get_level(m)+1)|
        nbr_node_id<-diff(my_nbrs(node_id,Links),{get_sender(m)})});
```

```

done :=
  if empty(to_be_sent) and (get_level(m)==att_filter_depth or
    empty(diff(my_nbrs(node_id,Links),{get_sender(m)})))
  then R.true
  else R.false

output sendUpdate1(next_update)
  pre next_update~=NULL
  eff to_be_sent := diff(to_be_sent,{next_update});
  next_update := NULL;
  done := if empty(diff(to_be_sent,{next_update})) then R.true else R.false

output sendUpdate2(next_update)
  pre next_update~=NULL
  eff to_be_sent := diff(to_be_sent,{next_update});
  next_update := NULL;
  done := if empty(diff(to_be_sent,{next_update})) then R.true else R.false

output doneUpdate(done)
  pre done=(R.true)
  eff done := R.false

begincsp

fst((a,b)) = a
snd((a,b)) = b

-- some functions to extract the contents of a message
get_sender(S_) = NULL
get_sender(U.n1.n2.h1.h2.1) = n1
get_receiver(S_) = NULL
get_receiver(U.n1.n2.h1.h2.1) = n2
get_hash1(S_) = NULL
get_hash1(U.n1.n2.h1.h2.1) = h1
get_hash2(S_) = NULL
get_hash2(U.n1.n2.h1.h2.1) = h2
get_hashes(S_) = NULL
get_hashes(U.n1.n2.h1.h2.1) = h1.h2
get_level(S_) = NULL
get_level(U.n1.n2.h1.h2.1) = 1

my_links(n,all_links) = {{i,n} | i <- MyNodes, member({i,n},all_links)}

my_nbrs(n,my_links) = {i | i <- MyNodes, member({i,n},my_links)}

-- a function that returns the nth element of a sequence (counting from 0)
nth(n,seq) = if n==0 then head(seq)
             else nth(n-1,tail(seq))

-- Below is an example of how we are representing attenuated Bloom filters,
-- i.e. a sequence of sequences of booleans:

-- <<true ,false,true ,false,false,true ,false,false>,
-- <false,false,false,true ,false,true ,true ,false>,
-- <false,true ,true ,true ,true ,false,false,true >>

-- The following function creates a new attenuated Bloom filter of width w and
-- depth d
NewAttFilter(1,1) = <<false>>
NewAttFilter(w,1) = <<false>>^head(NewAttFilter(w-1,1))>
NewAttFilter(w,d) = <head(NewAttFilter(w,d-1))>^NewAttFilter(w,d-1)

-- This function creates a new ordinary Bloom filter of width w

```

```

NewFilter(1) = <false>
NewFilter(w) = <false>^NewFilter(w-1)

-- A function that adds a 1 at place n of a Bloom filter
add_to_filter(0,filter) = <true>^tail(filter)
add_to_filter(n,filter) = <head(filter)>^add_to_filter(n-1,tail(filter))

-- This function adds two 1s to a Bloom filter in the places indicated by the
-- two-element sequence ns.
add_to_filter2(ns,filter) =
  add_to_filter(nth(0,ns),add_to_filter(nth(1,ns),filter))

-- A function that creates new attenuated Bloom filters associated with each
-- specified neighbour.
make_att_filters(nbrs) =
  {(n,NewAttFilter(filter_width,att_filter_depth)) | n <- nbrs}

-- A function that adds a 1 at place n at depth d of an attenuated Bloom
-- filter. Note that places in a filter start at 0, and filter depths start at
-- 1.
add_to_att_filter(n,1,att_filter) =
  <add_to_filter(n,head(att_filter))>^tail(att_filter)
add_to_att_filter(n,d,att_filter) =
  <head(att_filter)>^add_to_att_filter(n,d-1,tail(att_filter))

-- This function adds two 1s to an attenuated Bloom filter in the places
-- indicated by the two-element sequence ns, and at depth d.
add_to_att_filter2(ns,d,att_filter) =
  add_to_att_filter(nth(0,ns),d,add_to_att_filter(nth(1,ns),d,att_filter))

-- The following function updates in the above manner an attenuated Bloom
-- filter that is associated with the link to the given neighbour. The
-- link_att_filters parameter must be a set of (neighbour,attenuated Bloom
-- filter) pairs.
add_to_link_att_filters2(ns,d,nbr,link_att_filters) =
  let
    pair = {m | m <- link_att_filters, nbr==fst(m)}
    att_filter = {snd(m) | m <- link_att_filters, nbr==fst(m)}
  within
    union(diff(link_att_filters,pair),{(nbr,add_to_att_filter2(ns,d,Pick(att_filter)))})

-- A function that creates the initial set of updates that a node needs to send
-- out.
make_updates(node_id,h1,h2,data,nbrs) =
  {U.node_id.n.nth(d,h1).nth(d,h2).1 | d <- data, n <- nbrs}

-- This function returns a set containing the possible pairs of values
-- resulting from applying the two hash functions to a piece of data.
possible_pairs_of_hashes(<>,<>) = {}
possible_pairs_of_hashes(h1,h2) =
  union({head(h1).head(h2)},possible_pairs_of_hashes(tail(h1),tail(h2)))

endcsp

```

## A.2 Query Algorithm

automaton NODE\_QUERY

```

signature
  input  newQuery(m: {|D|})
  input  receiveQuery1(m: {|Q|})
  input  receiveQuery2(m: {|Q|})
  output sendQuery1(m: {|Q|})

```

```

output sendQuery2(m: {|Q|})
output outcome(m: {|R|})

states
  final buffer_limit : Int,
  final data : Set(Data),
  final hash1 : HashFn,
  final hash2 : HashFn,
  final link_att_filters : Set((MyNodes,Seq(Seq(Bool))))),
  query_buffer : Seq(AnyMsg) := <>,
  result : AnyMsg := NULL

transitions
input newQuery(m)
  pre #query_buffer~=buffer_limit;
    member(get_name(m),diff(Data,data));
    get_id(m)=node_id
  eff query_buffer :=
    if empty({i|i<-my_nbrs(node_id,Links),att_filter_matches(nth(get_name(m),hash1),
      nth(get_name(m),hash2),get_att_filter(i,link_att_filters))})
    then query_buffer
    else query_buffer^<Q.node_id.Min({i|i<-my_nbrs(node_id,Links),
      att_filter_matches(nth(get_name(m),hash1),nth(get_name(m),hash2),
      get_att_filter(i,link_att_filters))}).get_name(m).nth(get_name(m),hash1).
      nth(get_name(m),hash2).{node_id}>
  result :=
    if empty({i|i<-my_nbrs(node_id,Links),att_filter_matches(nth(get_name(m),hash1),
      nth(get_name(m),hash2),get_att_filter(i,link_att_filters))})
    then R.node_id.get_name(m).false
    else NULL

input receiveQuery1(m)
  pre get_sender(m)~=node_id;
    get_receiver(m)=node_id;
    not member(node_id,get_visited(m));
    member(get_sender(m),get_visited(m));
    member(get_hashes(m),possible_pairs_of_hashes(hash1,hash2))
  eff query_buffer :=
    if member(get_name(m),data) or #query_buffer==buffer_limit or
      empty(diff({i|i<-my_nbrs(node_id,Links),att_filter_matches(get_hash1(m),
      get_hash2(m),get_att_filter(i,link_att_filters))},get_visited(m)))
    then query_buffer
    else query_buffer^<Q.node_id.Min(diff({i|i<-my_nbrs(node_id,Links),
      att_filter_matches(get_hash1(m),get_hash2(m),
      get_att_filter(i,link_att_filters))},get_visited(m))).get_name(m).
      get_hash1(m).get_hash2(m).union(get_visited(m),{node_id}>;
  result :=
    if member(get_name(m),data)
    then R.node_id.get_name(m).true
    else if #query_buffer==buffer_limit or empty(diff({i|i<-my_nbrs(node_id,Links),
      att_filter_matches(get_hash1(m),get_hash2(m),
      get_att_filter(i,link_att_filters))},get_visited(m)))
    then R.node_id.get_name(m).false
    else NULL

input receiveQuery2(m)
  pre get_sender(m)~=node_id;
    get_receiver(m)=node_id;
    not member(node_id,get_visited(m));
    member(get_sender(m),get_visited(m));
    member(get_hashes(m),possible_pairs_of_hashes(hash1,hash2))
  eff query_buffer :=
    if member(get_name(m),data) or #query_buffer==buffer_limit or
      empty(diff({i|i<-my_nbrs(node_id,Links),att_filter_matches(get_hash1(m),

```

```

        get_hash2(m),get_att_filter(i,link_att_filters))},get_visited(m))
    then query_buffer
  else query_buffer^<Q.node_id.Min(diff({i|i<-my_nbrs(node_id,Links),
    att_filter_matches(get_hash1(m),get_hash2(m),
    get_att_filter(i,link_att_filters))},get_visited(m))).get_name(m).
    get_hash1(m).get_hash2(m).union(get_visited(m),{node_id})>;
result :=
  if member(get_name(m),data)
  then R.node_id.get_name(m).true
  else if #query_buffer==buffer_limit or empty(diff({i|i<-my_nbrs(node_id,Links),
    att_filter_matches(get_hash1(m),get_hash2(m),
    get_att_filter(i,link_att_filters))},get_visited(m)))
    then R.node_id.get_name(m).false
    else NULL

output sendQuery1(head(query_buffer))
  pre query_buffer~=<>
  eff query_buffer := tail(query_buffer)

output sendQuery2(head(query_buffer))
  pre query_buffer~=<>
  eff query_buffer := tail(query_buffer)

output outcome(result)
  pre result~=NULL
  eff result := NULL

begincsp

fst((a,b)) = a
snd((a,b)) = b

-- some functions to extract the contents of a message
get_id(D.me.d) = me
get_sender(Q.n1.n2.d.h1.h2.v) = n1
get_receiver(Q.n1.n2.d.h1.h2.v) = n2
get_name(D.me.d) = d
get_name(Q.n1.n2.d.h1.h2.v) = d
get_hash1(Q.n1.n2.d.h1.h2.v) = h1
get_hash2(Q.n1.n2.d.h1.h2.v) = h2
get_hashes(Q.n1.n2.d.h1.h2.v) = h1.h2
get_visited(Q.n1.n2.d.h1.h2.v) = v

my_links(n,all_links) = {i,n} | i <- MyNodes, member({i,n},all_links)}

my_nbrs(n,my_links) = {i | i <- MyNodes, member({i,n},my_links)}

-- a function that returns the nth element of a sequence (counting from 0)
nth(n,seq) = if n==0 then head(seq)
             else nth(n-1,tail(seq))

-- This function returns true if the two hash values are both present at any of
-- the levels of the given attenuated Bloom filter.
att_filter_matches(h1,h2,<>) = false
att_filter_matches(h1,h2,att_filter) =
  (nth(h1,head(att_filter)) and nth(h2,head(att_filter)))
  or
  att_filter_matches(h1,h2,tail(att_filter))

-- This function returns a set containing the possible pairs of values
-- resulting from applying the two hash functions to a piece of data.
possible_pairs_of_hashes(<>,<>) = {}
possible_pairs_of_hashes(h1,h2) =
  union({head(h1).head(h2)},possible_pairs_of_hashes(tail(h1),tail(h2)))

```

```
-- a function that extracts the attenuated Bloom filter corresponding to  
-- neighbour i from the given set of (neighbour,attenuated Bloom filter) pairs  
get_att_filter(i,link_att_filters) = Pick({snd(k)|k<-link_att_filters,i==fst(k)})  
  
endcsp
```

## References

- [1] Dia homepage. <http://www.gnome.org/projects/dia/>.
- [2] D. Aguayo, D. De Couto, W. Lin, H. Lee, and J. Li. Grid: Building a robust ad hoc network. In *Proceedings of the 2001 Student Oxygen Workshop*. MIT Laboratory for Computer Science, 2001.
- [3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [4] Sadie Creese. *Data independent induction: CSP Model Checking of Arbitrary Sized Networks*, 2001. D.Phil. Thesis, University of Oxford.
- [5] S. J. Garland, N. A. Lynch, and M. Vaziri. IOA: A language for specifying, programming, and validating distributed systems, 1997.
- [6] John Kubiawicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.
- [7] Sean C. Rhea and John Kubiawicz. Probabilistic location and routing. In *Proceedings of INFOCOM 2002*, 2002.
- [8] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.
- [9] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.