A State Machine-Based Approach For Reliable Adaptive Distributed Systems

Leonardo Mostarda, Daniel Sykes, Naranker Dulay
Department of Computing
Imperial College London
Email: {lmostard, daniel.sykes, nd}@imperial.ac.uk

***Abstract*—Adaptive systems are often composed of distributed components that co-operate in order to achieve a global behaviour, and yet many approaches for adaptive systems are centralised or make strong assumptions about the distributed aspects of the problem. However, if insufficient attention is paid to the problem of decentralisation, especially in the difficult and unpredictable environments in which adaptive systems are commonly deployed, it can introduce inefficiencies, and even cause catastrophic failure. An adaptive system is either required to implement subtle synchronisation and consensus protocols or accept certain types of failure from which the system cannot recover. A major goal of our research is to facilitate the development of adaptive, reliable and distributed applications. We provide a framework in which a state machine language is used to define logically centralised behaviour. This is automatically translated into a reliable and efficient distributed implementation that enforces the correct co-ordination in the presence of unpredictable failures.**

***Keywords*-Adaptive systems, fault tolerance, distributed systems.**

I. Introduction

Adaptive systems have been proposed as a solution for enabling systems to continue meeting their requirements or goals in the face of unpredictable variation in the execution environment [1]. In a centralised system, other applications can contend for memory and CPU attention; in a distributed system, network bandwidth and connectivity is variable; in an embedded or ubiquitous system, properties of the physical environment change without regard for the stability of applications. These environments can impact a system's performance, reliability and indeed its very capacity to perform the function it was designed for.

Such change is naturally unpredictable and attempts to pre-empt it rely on some form of closed-world assumption. A better approach is to endow the system with adaptive techniques which observe the actual runtime conditions and make decisions about how to change the application behaviour so that the requirements can be upheld, creating a feedback loop.

Much existing work on adaptive systems [2], [3] makes the strong assumption that the adaptive controller is reliable and can co-ordinate any distributed application components (insofar as they exist) without faults. In other words, such systems are centralised. For example, Rainbow [2] can apply centralised repair strategies to client-server applications. However, in many of the application areas of adaptive systems, especially embedded systems, distribution is a common requirement. If reliable distribution is assumed rather than addressed, the adaptive controller itself risks introducing another point of inefficiency and failure, indeed one from which it is unlikely to be able to recover.

In this work, we seek to provide a framework, on which an adaptive system can be based, which handles the reliable distribution of the adaptive behaviour without encumbering the programmer with the manifold concerns introduced by decentralisation. Many existing centralised adaptive techniques can be adjusted to work on our platform, providing distribution in exchange for minimal effort.

Our platform, called GOANNA [4], uses finite state machines (FSMs) as a formal basis for co-ordinating a set of distributed application components. Adaptive algorithms specified using our FSM language can be distributed automatically by the platform which ensures correct execution and fault tolerance by employing a consensus protocol based on Multi-Paxos [5]. Additionally, components are grouped into *sets* which provides an extra dimension of robustness as individual components are permitted to join and leave the system at runtime without affecting the correct execution of the adaptive behaviour.

The main contributions of this work, then, are:

- A framework for specifying adaptive algorithms which can then be distributed to provide a reliable decentralised implementation. A key benefit is the separation of the adaptive and the distributed concerns. The use of finite state machines also aids in the independent verification of the adaptive algorithm, although we do not specifically address this here.
- A novel fault-tolerant consensus protocol for distributed state machine execution based upon Multi-Paxos.

In Section II we introduce an example adaptive system which will be used to explain aspects of our approach. Section III gives an overview of the approach and Section IV describes our language for defining state machines. Section V describes the state machine decomposition process and the consensus protocol, the performance of which is discussed in Section VI. Finally we comment on related work and conclude.

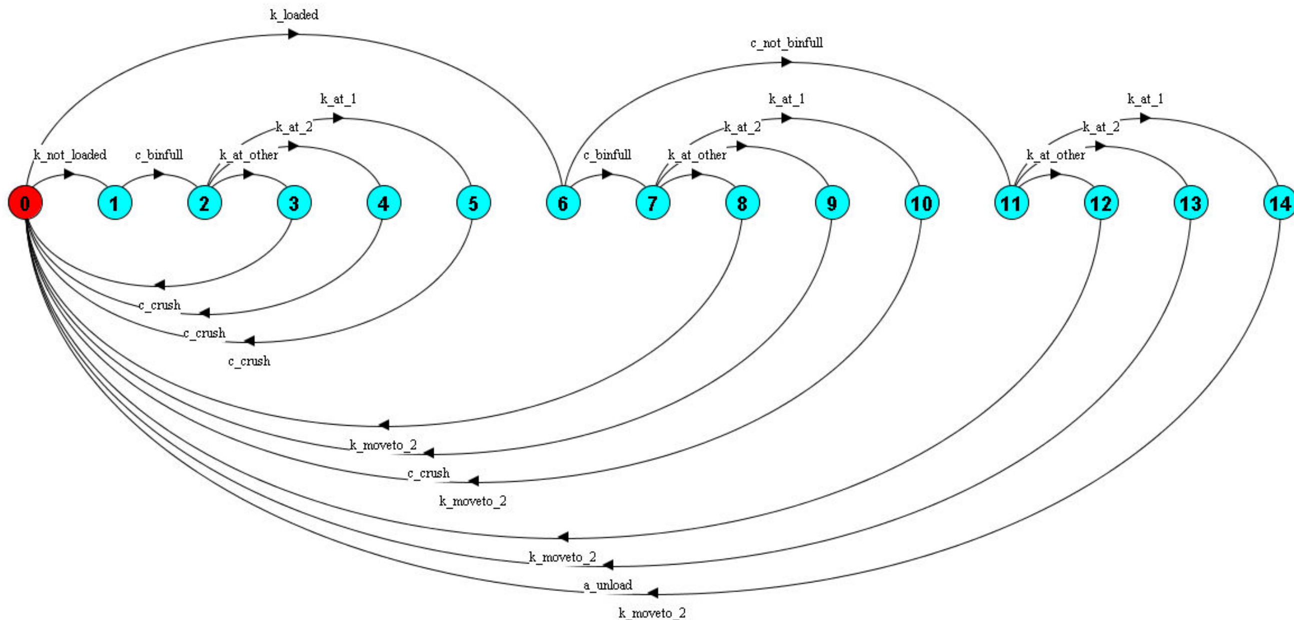

Figure 1. The complete state machine. A label such as `k_moveto_2` refers to the action $moveToLocation2$ of the $Koala$ component. A label such as `c_not_binfull` refers to the event in which the $Crusher$ observes that the bin is not full.

II. Case Study

To illustrate our approach, we refer to a small embedded application that co-ordinates several mobile robots to achieve a global goal in the face of an uncertain environment. The global behaviour is expressed as a *reactive plan* [6] which handles unexpected transitions in the observable state of the environment by continually sensing and choosing actions which lead from the sensed state to the satisfaction of the goal. In this manner, the plan adapts to the unpredictable evolution of the environment. The plan is generated from a description of the capabilities of each robot with respect to a model of the environment, and an abstract goal denoting the state, or states, to which the system should progress (although in this paper we are not concerned with the mechanics of plan generation). This technique is particularly appropriate for our approach as plans can be easily expressed in the form of a state machine.

For our example application, there are three small robots: a *Koala* which can carry balls (representing waste materials), a fixed *Arm* which can pick and place balls, and a *Crusher* which destroys them. The behaviour of the system as a whole is to collect balls from a room, and have them crushed (to clean the room). The plan is generated using the available application-dependent actions such as $moveToLocation1$ (start moving to location 1), $moveToLocation2$, and *unload* (the ball), providing a set of paths (traces) such as those in Figure 1. For instance the transitions between states 0, 6, 11, 14 and 0 specify that when the Koala has been loaded with a ball, the bin (in front of the crusher) is not full and the Koala is at location 1, then the Koala should move to location 2 (next to the arm). Notice that the act of moving to location 2 does not guarantee that in the next step the environment will be in a state where the Koala is at location 2. An unmodelled portion of the environment (such as an unwitting human) may interact to return the Koala to location 1, for example. In this case, the plan can continue by attempting to move again. If the action succeeds, then the transitions between states 0, 6, 11, 13 and 0 specify that when the Koala has moved to location 2 the arm can load the ball.

If this system were implemented in a centralised manner, a failure in the node controlling plan execution would be catastrophic. Additionally, the plan execution middleware would be forced to implement appropriate algorithms for handling lost connectivity or failed robots. Our platform encapsulates these distribution concerns from any adaptive approach which is expressible in our FSM language, and removes the reliance on the central node.

III. Approach overview

In the GOANNA approach, we assume the system is composed of a set of components which provide and require services. These components are then co-ordinated to achieve a global adaptive behaviour via the specification of a (global) finite state machine. This state machine specifies the sequences of events (resulting from service invocations) which are permitted in the running system.

To achieve a robust decentralisation of the global state machine, it is decomposed into a set of local state machines.

A *leader* node (which uses a skeleton as described in Section V-A) is used in order to ensure that the local state machines implement the global FSM. Thus the distributed implementation has the same behaviour as the centralised specification. This is achieved by using our consensus protocol that extends Multi-Paxos and is described in Section V-B.

If the leader fails, the protocol handles the election of a new one. The decomposition process is described in Section V-A.

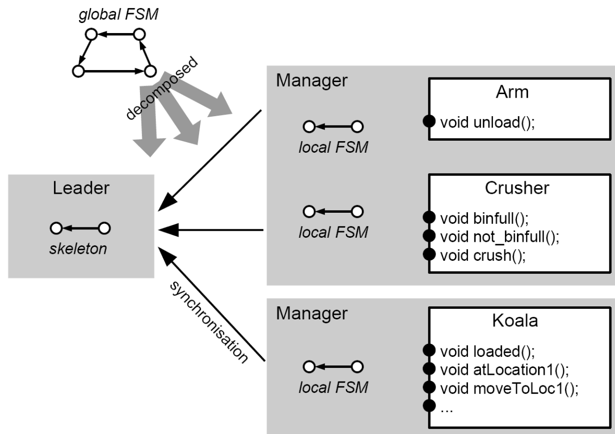

Figure 2. GOANNA overview.

Figure 2 shows three components from our case study which provide various services. The GOANNA platform gives to each host a *manager* which handles the consensus protocol and stores the local state machines. In this example there are Arm and Crusher components associated with the same manager. This manager contains the local state machines pertaining to the Arm and the Crusher. The Koala component is associated with another manager. Each manager changes the FSM state in response to service invocations on each component and the leader checks that these state changes are in accordance with the global behaviour. Managers may also invoke component services. For example, after a sequence of events corresponding to the Koala being loaded, the crusher's bin being empty and the Koala being at location 1, the Koala manager invokes the $moveToLocation2$ service. Note that the environmental events (such as $loaded$) are provided services of each component. Intuitively, these are called by the environment itself (or an appropriate sensor component), at which point the corresponding manager can respond.

In the next section, we describe the language used to define global FSMs, before describing the technical details of the decomposition and consensus algorithms.

```
1  global fsm globalPlan(set Crusher crusherSet,
2    set Arm armSet,set Koala koalaSet){
3  loaded on koalaSet from *
4    0-6: -> {}
5  not_binfull on crusherSet from *
6    6-11:  -> {}
7  load on armSet from *
8    13-0:  -> {}
9  atLocation1 on koalaSet from *
10   11-14:  -> {moveToLocation2();}
11 moveToLocation2 on koalaSet from *
12   14-0:  -> {}
13 atLocation2 on koalaSet from *
14   11-13:  -> {load();}
15 on timeout(20000)
16   0-0:->{alert();}
17 ...
18 }
```

Figure 3. The (partial) state machine description.

IV. State machine language

A global state machine in our language consists of a list of *event-state-condition-action* rules defined in terms of the participating components (grouped into sets as described below). Figure 3 shows the state machine for our case study. Each rule states that when the system is in the given state and the event is observed, then if the condition holds, the system should perform the stated action.

For example, the fourth rule in Figure 3 states that when the state is 11 and the event $atLocation1$ is observed on a Koala, then (since the condition is empty), the action $moveToLocation2$ is performed on the same component, moving the state machine to state 14.

A. Events

Four kinds of events can be captured in a GOANNA state machine. On the client side, outgoing invocations and returned responses are captured. On the server side, incoming invocations and outgoing responses are captured. Mapping service invocations into four different events provides flexibility, since co-ordination can be defined using only client side events, only server side events, or both, as needed. For instance in Figure 3 the event "`loaded on koalaSet from *`" corresponds to a `loaded` incoming service call observed on a Koala. In this case we do not specify the client that performs the invocation, indicated by '`*`'

Timeout events are also supported. Timeout events are generated by the leader when no rule has been applied within the specified time t. For example, `timeout(20000)` will raise a timeout after 20 seconds if no other rule has been applied.

B. States, conditions, actions

For each event as described above the state machine can define a list of state-condition-action tuples. A state-condition-action is of the form q_s-q_d: $\{$ *condition* $\} \rightarrow \{$ *action* $\}$ where q_s and q_d are the start and end states, while

the condition and action are a predicate and a piece of code respectively. When an event is observed, the current state of the global state machine is q_s, and the condition is satisfied, then the action can be executed and the current state is set to the end state q_d. Note that the first state listed in the state machine definition is assumed to be the initial state for the FSM. For example in Figure 3 the initial state is 0.

If an event is observed, but the rule cannot be applied (if the condition does not hold, or there is no relevant transition from the current state), then one of several policies, such as $retry$ (the event acceptance) or $discard$ (the event) can be enforced.

C. Sets

The state machine is given in terms of sets of components which group together component instances of the same type. For instance Figure 3 specifies the state machine `globalPlan` which is parameterised by `crushSet`, `armSet` and `koalaSet`, which are sets that group together components of the type `Crusher`, `Arm` and `Koala`, respectively.

The set parameters are defined when the state machine is instantiated in a separate *configuration specification*. Each set is defined using a component type[1] and optionally a further `where` predicate that can use attributes such as host name, position, and node capabilities to group components as they are discovered. For example, in Figure 4 there are three set definitions: `a`, `k` and `c`. The set `a` groups components of the type `Arm` that reside in location 2. `k` includes all `Koala` components deployed in the system, and `c` defines a `Crusher` component that runs on a specific host `crusherHost`.

```
1 configuration RobotApplication {
2     set a:Arm where (location=="location2");
3     set k:Koala where (host==ALL);
4     set c:Crusher where (host=="crusherHost");
5
6     instance arm1:globalPlan(a,k,c);
7 }
```

Figure 4. A configuration definition.

Components can join and leave sets at runtime. When an action of the global FSM must be performed an instance from the appropriate set is selected. This isolates the management of component availability from the state machine specification and allows the selection of new available components in case of failure.

Sets are maintained by the leader node which has a registry of all manager addresses, and the sets to which managers belong.

A configuration can instantiate multiple global FSMs for different applications. However, these do not interact and in much of the following we assume a single global FSM instance.

D. Semantics

While the informal description of our state machines provides an intuition, in the following we provide a formal description of the acceptance criterion and the language accepted by them. We first provide some definitions. The set E denotes the set of all possible component events while e_1, e_2 ... e_n are elements in E. The set E^c denotes the set of events locally observed on a component c and e_1^c ... e_n^c elements in E^c.

Definition 1: Let $S = \{c_1, \ldots, c_i, \ldots\}$ be a system where each c_i is a component instance. A trace $t = e_1, \ldots, e_i, \ldots$ of S is a sequence of events in E. We denote with T_S the set of all possible traces in the system S. The traces in T_S are subject to the causality relation presented in [7] defining a partial ordering on the events in the distributed system.

Definition 2: A state machine is 4-tuple $A = (Q, q_0, I, rules)$ where: (i) Q is a finite set of states; (ii) $q_0 \in Q$ is the initial state; (iii) I is a finite set of events s.t. $I \subseteq E$; and (iv) rules is a list of 5-tuples (e,q_s,q_d,condition,action) where $e \in E$ and $q_s, q_d \in Q$.

Definition 3: Let $A = (Q, q_0, I, rules)$ be a state machine and $e \in I$ be an event. Let q be the current state of A. The event e *can be accepted* by a rule (e,q_s,q_d,condition,action) in $rules$ if $q = q_s$ and the condition is satisfied.

Definition 4: Let $A = (Q, q_0, I, rules)$ be a state machine and $t = e_1 \ldots e_i \ldots$ a trace in T_S. Let q_0 be the initial state of A and e_1 be the first symbol to read. A accepts the sequence t if for each current state q_{i-1} and next symbol e_i, A can accept e_i by a rule (e_i,q_{i-1},q_i,condition,action). When the rule is applied the action is performed, q_i is the new state of A and e_{i+1} the next symbol to read.

Definition 5: The language T_A recognised by a state machine A is composed of all traces accepted by it.

When multiple global state machines are defined the event must be accepted by all of them.

Note that the set T_A is a subset of T_S since the components can, at runtime, produce events that cannot be accepted by the state machine. In other words, if a trace $t_s \in T_S$ and a trace $t_a \in T_A$ move S and A, respectively, from state s to s', then t_a can be derived from t_s by deleting those events which have not been accepted, but where instead a reaction policy such as $retry$ (the parsing) or $discard$ (the event) has been applied.

In the following we introduce notations and definitions related to our global state machine decomposition process:

- M denotes the set of all managers and m_1, ..., m_n are elements in M;
- G denotes the set of all global state machine *instances*[2]

[1]The component type must be always specified to allow a compile-time check that the services and events mentioned in the state machine are in fact provided by the components.

[2]State machine instances are names that identify FSM instances defined inside the configuration file.

as defined in the configuration files. $\underline{A_1}$, ..., $\underline{A_i}$ are elements in G and A_1,..., A_i the corresponding global FSM definition;

- S denotes all set definitions as defined in the configuration file, s^c denotes a set in S whose definition is based on a component type c.
- LS denotes the set of all local FSMs and $A_s \in LS$ denotes a local FSM related to the set s.
- K denotes the set of all component instances and K_m denotes the set of instances local to the manager $m \in M$.

Definition 6: A *manager* $m \in M$ is a pair (K_m, fm) where fm is a function $fm : G \rightarrow \mathbb{Z} \times 2^{LS}$. The function fm relates each global FSM $\underline{A_i}$ to an integer in $\mathbb{Z}$ denoting its state (the last updated state the manager is aware of) and to a set of local state machines derived from A_i. More specifically, fm relates A_{s^c} to $\underline{A_i}$ iff component c is a member of K_m, (i.e., the set s^c has been allocated by the manager m).

Definition 7: The *leader* L is a pair (fs, fl) where fs is a function $fs : S \rightarrow 2^M$ that relates to each set s all managers where the set has been allocated, fl is a function $fl : G \rightarrow \mathbb{Z} \times A_k$ that relates to each global FSM $\underline{A}$ its state and the corresponding skeleton A_k.

V. DISTRIBUTION

In the following we first introduce the global state machine decomposition process then we give the details of our consensus protocol.

A. State machine decomposition

We decompose each state machine A into a set of local ones plus a skeleton. More specifically if A is defined over the sets s_1, ..., s_n our decomposition process generates a set of local state machines A_{s_1}, ..., A_{s_n}.

Let $A = (Q, q_0, I, rules)$ be a global FSM that is defined over a set s^c. In the following we show how to generate the local state machine $A_{s^c} = (Q_{s^c}, q_{s0}, I_{s^c}, rules_{s^c})$ and the skeleton $A_k = (Q_k, q_{k0}, I_k, rules_k)$.

We generate the local state machine A_{s^c} by examining the global state machine A for rules of the form $R = (e^c, q_s, q_d, condition, action)$ where e^c is an event observed on component type c. Every time one of these rules is found, the event e^c is added to I_{s^c}, the states q_s and q_d are added to Q_{s^c} and the rule R is added to $rules_{s^c}$. In other words the state machine A_{s^c} contains all interactions that take place locally on a component belonging to the set s^c. In case q_s or q_d is the initial state then $q_0 = q_s$ or $q_0 = q_d$, respectively. When the initial state is not specified a random one can be chosen. This does not affect the correctness of the approach since the leader always synchronises the manager with the new correct state (see next section for details).

Figure 5 shows the local state machine generated for the `koalaSet` set. This is defined over the component type

```
1  local koalaSet  fsm globalPlan(set Crusher crusherSet,
2    set Arm armSet,set Koala koalaSet){
3  loaded on koalaSet from *
4    0-6: -> {}
5  atLocation1 on koalaSet from *
6    11-14:  -> {moveToLocation2();}
7  moveToLocation2 on koalaSet from *
8    14-0:  -> {}
9  atLocation2 on koalaSet from *
10   11-13:  -> {load();}
11 ...
12 }
```

Figure 5. The automatically-generated local state machine for the Koala.

Koala therefore each rule of the local state machine refers to an event observed on the Koala (such as $atLocation1$).

In order to generate the leader skeleton A_k we examine the global state machine for rules pertaining to timeout events and add them to the rules $rules_k$ of the skeleton. In other words the skeleton contains all timeout rules acting as a global timer.

B. GOANNA consensus protocol

Our consensus protocol extends Multi-Paxos with Steady State with additional information in order to have a correct distributed state machine implementation. Multi-Paxos uses a leader to ensure progress and has improved performance [5], [8]. Our protocol is an extension that adds the information needed to execute actions and parse local event traces correctly. In particular we add timeouts to manage the one-to-one communications between managers executing an action and the leader checking it.

Multi-Paxos is normally described using client, acceptor, learner and leader[3] roles. In GOANNA, the client, acceptor and learner roles are merged into the manager role. A manager uses its old state to verify the event acceptance locally before proposing its new state. After a new state proposal the leader can either decline the request (e.g., the manager's state can be out of date) or accept it, waiting for the action to complete and the new state to be updated. Although these steps are the basis for correct distribution they are not efficient in terms of memory and traffic overhead. State machines can be composed of millions of states [9] so their deployment on each host can be inefficient. Moreover, managers could continuously propose their new local states overloading the network. Our global state machine distribution process offers a partition of the transitions and are loaded only when needed. While a consensus protocol solves the general problem of agreement between entities, in our approach we can take advantage of the state machine structure in order to avoid useless communication. The idea is that an outdated local state can be enough to reject an event (i.e, the states do not always need to be updated). More specifically a manager can reject an event that cannot

[3]The leader is also known as the proposer.

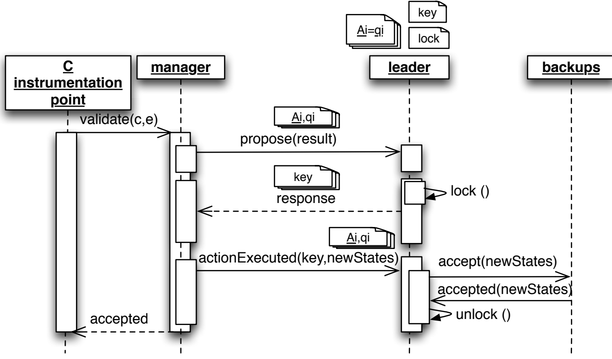

Figure 6. Successful protocol execution.

be accepted in its current outdated state q and in any state reachable from q.

1) Successful protocol execution: The local state machines and the leader's skeleton are used to implement the global state machine instance(s) in a distributed way. In order to ease the presentation we assume only one FSM instance $\underline{A_i}$ of the type $A = (Q, q_0, I, rules)$. The general case is a straightforward extension.

In Figure 6 we show a successful protocol execution. The protocol starts when an instrumentation point related to a component instance c detects an incoming/outgoing message. The instrumentation point generates a component event e and invokes the procedure `validate(c,e)` on its local manager. The procedure uses the last updated state q_i of $\underline{A_i}$ to verify the acceptance of event e. More specifically, the procedure considers each set s which the component instance c belongs to and tries to find a rule q_i-q_d: $\{$ *condition* $\} \rightarrow \{$ *action* $\}$ of A_s that can be applied (see Definition 3). If the rule is found the manager starts the protocol by sending a `propose(result)` request to the leader containing the machine instance $\underline{A_i}$ and its proposed state q_i. The leader receives the request and compares the received state q_i with the one of its instance $\underline{A_i}$, e.g., $\underline{q_i}$. Moreover it checks whether or not the instance has been locked by another manager. Suppose that the states are the same and the instance has not been locked. Then the leader generates a new protocol key and responds with a `response` data structure to the manager where `response.key` and `response.outcome` are set to the new key and the constant `accepted`, respectively. With this answer the leader promises to the manager the lock on the required FSM instance. The manager receives the request, performs the local actions (from the rules), and sends back to the leader an $actionExecuted(response.key, newStates)$ response where $newStates$ is the new state after the execution of the rule (q_d if the aforementioned rule is applied). The leader receives the request and checks the existence of the key. In case the key exists it deletes the key, unlocks the instance and updates its local state with the received one. The process of updating the state requires the leader to perform a Multi-Paxos protocol with Steady State. More specifically, the new state is sent to a set of backup managers through an `accept` request. When the majority of them notify the update (through an `accepted` request) the protocol can terminate correctly. As we will show in the following this ensures the replication of the state values across several managers to perform a reliable leader election after leader faults.

When multiple state machine instances are defined the manager must check the event acceptance for all of them (see Section IV for the acceptance criterion). As for the aforementioned execution if the event is accepted the manager starts the protocol but communicates all the states, locks all state machine instances and applies all actions (when it receives the grant from the leader). We also mention that the leader can change state without any manager interaction. This occurs when timeout rules, as specified in the state machines, are applied.

A protocol execution can raise different exceptions as a consequence of link failures, node failures and so on. In the next section we show how our protocol handles those failures.

2) Protocol exceptions: A protocol instance can raise *manager out-of-sync* and *FSM locked* exceptions. A *manager out-of-sync* exception (Figure 7) is raised when any of the states sent by the manager are different from the state of the global FSM on the leader. This is a consequence of a manager whose proposed states are not synchronised with the global execution and is detected and notified by the leader. In particular after the leader receives the request $propose(result)$ it replies with a `response` data structure containing the following information: (i) the field `response.outcome` set to `out_of_sync`; (ii) a list of tuples $(\underline{A_i}, \underline{q_i})$ containing the state machine instances and the correct global states. These updated global states can be used to parse again the event.

A *locked* exception is generated when a manager proposes a list of tuples $(\underline{A_i}, \underline{q_i})$ but one of the instances (e.g., $\underline{A_i}$) has already been locked by another manager. In this case the leader sends back a `response` data structure where the field `response.outcome` is set to `locked`.

Failures of managers and communication links are handled in our protocol by using timeouts. In the following we describe those failures and how they are handled by the leader and by managers.

A leader can see a manager or link failure during three possible steps of the protocol execution: (i) when it is responding to a `propose` request (*propose response failure*); (ii) while waiting for an `actionExecution` message (*action execution timeout*); (iii) when responding to an `actionExecution` message (*action execution response failure*). These faults can be a result of a manager fault, a

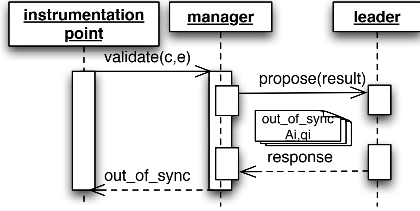

Figure 7. FSM manager out-of-sync handling.

communication failure or a slow (overloaded) manager.

- A *propose response failure* occurs when the leader fails to communicate to a manager the outcome of a proposal of states (i.e, a `response` data structure). In this case a timeout is raised and the leader deletes any key or lock granted.
- An *action execution timeout* occurs when a manager receives the permission to execute its local actions but it does not respond with an $actionExecuted$ message. In this case the timeout is triggered on the leader side. This causes the key to be deleted (i.e., the protocol instance to be ended) and all instances to be unlocked. It is worth mentioning that even if the manager sends an $actionExecuted$ invocation after the timeout expires this will be detected (the key no longer exists) and the states on the leader will not be updated. Therefore in the case of non-recoverable actions the global execution can be inconsistent.
  The *action execution timeout* provides resilience to component faults. When one component fails to execute its action the leader does not update the states of the state machine instances (that is, the global behaviour did not progress), it times out and waits for a new request. In this way a new component instance (correctly synchronised) can still perform another action.
- An *action execution response failure* occurs when the leader correctly receives an `actionExecution` message from a manager but fails to acknowledge the reception. In this case the leader ends the protocol instance and waits for the next request.

A manager can see a leader or link failure during four possible steps of the protocol execution: (i) when invoking to a propose request (*propose invocation failure*); (ii) while waiting after the propose request (*propose response failure*); (iii) when invoking the actionExecution (*action execution invocation failure*); (iv) while waiting the actionExecution response (*actionExecution response failure*). These faults can be a result of a leader fault, a communication failure or slow leader execution.

- *A propose invocation failure* occurs when a manager fails to contact the leader at the beginning of the protocol. In this case no instance of the protocol is started and an error is returned to the instrumentation point.
- *A propose response failure* occurs when a manager correctly proposes the new states to the leader but it does not receive an answer. In this case the manager ends the protocol execution and ignores any successive message related to the same protocol instance.
- *An action execution invocation failure* occurs when a manager cannot send the action execution acknowledgement to the leader, while *an action execution response failure* occurs when a manager correctly sent the action execution message to the leader but it does not receive an acknowledgement. In both cases the manager ends the protocol execution and ignores any successive message related to the same protocol instance. We emphasise that in the case of non-recoverable actions that the global execution can be inconsistent.

Protocol message	Bytes	Time (ms)
propose	4x(FSM instance number)	13
response	2+4x(FSM instance number)	13
actionExecuted	2+4x(FSM instance number)	13
actionExecuted ACK	2	13

Figure 8. Protocol overhead.

In our protocol, managers are entrusted to detect a leader failure. More specifically when the leader is no longer available managers detect it, a new leader is elected and all correct global states recovered from the backup managers[4].

One should be aware that there are cases in which the protocol may not make any progress. For instance this is the case in which the same manager is always granted permission and always fails. In order to avoid this kind of livelock the leader always chooses a random manager when granting permission.

VI. PERFORMANCE

In this section we present the space and time overhead for our implementation in order to demonstrate its efficiency and scalability. We have tested our case study components and used GOANNA for Java 1.5 version running on a 100 Mbit network of 60 Pentium IV 3.2 GHz machines each with 2 GB of RAM running the Linux operating system

A. Memory Overhead

A manager performs three main functions: (i) checks conditions; (ii) executes our consensus protocol; (iii) executes state machine actions. Functions (i) and (iii) can be arbitrary code, but are typically simple boolean expressions or calls to services. Successful execution of the consensus protocol requires four message exchanges between a manager and the leader. The overhead of the exchanged messages are

[4]We assume that after a fault the leader always stops its execution (this avoids multiple leaders). Nevertheless, multiple leaders executing at the same time are detected and the conflict is solved.

Process	Components	Heap (KB)	JVM (MB)
leader	10	317	10
leader	50	476	11
leader	100	499	13
manager	10	698	11
manager	50	775	13
manager	100	916	14

Figure 9. Leader and FSM Manager memory consumption.

FSM	Size (KB)
global fsm	3.8
Koala	2.5
Arm	0.9
Crusher	1.1

Figure 10. State machine file size.

summarised in the table of Figure 8 and are negligible. More specifically, in the worst case a message requires 4 bytes for each state machine instance (2 bytes for the state and 2 bytes for the FSM ID) plus 2 bytes for the protocol return code (see Section V-B1 for details). For instance in our case study a protocol message requires 6 bytes since we have 1 single state machine instance.

Table of Figure 9 summarises the memory costs of the manager and leader. More specifically, we have run one single leader and a single manager on two separate hosts. We have tested from 10 up to 100 Koala components plus one Arm and one Crusher connected to the same manager. In the worst case, i.e., 100 components running, the manager and leader memory (both heap and data) is 499KB and 916KB, respectively[5]. Thus our approach is appropriate for small constrained devices.

Table of Figure 10 shows the sizes of global FSM and each of the generated local FSMs for our case study. The sizes correspond to sizes of the serialised object for each FSM.

B. Execution Overhead

In order to study the overall performance of our distributed implementation, we looked at the time it takes for a manager to *validate* a component interaction event (i.e., the response time).

In order to evaluate the manager response time we have performed two experiments. In the first experiment we have the following set: (i) one Arm component; (ii) one Crusher component; (iii) from 1 to 60 Koala components; (iv) each component runs on a different machine and it is connected to a local manager; (iv) each component generates random requests every 500 ms; (v) the total simulation time is 60s. Effectively we have up to 62 managers running on different machines and a single leader. The table of Figure 11 shows the results obtained. As we can see the time remains constant as the number of managers increases. This is a consequence of the small amount of information required to perform the protocol (a few bytes) and the small total amount of components (62 at most). In the second experiment we increase the total number of components (i.e., robots) in the system by using the following settings: (i) a manager with exactly one Arm component connected to it; (ii) a manager with exactly one Crusher component connected to it; (iii) from 10 up to 60 managers running on different hosts where 60 Koala components are connected to each of them; (iv) each component generates random requests every 500 ms; (v) the total simulation time is 60s. In the table of Figure 12 we show the results obtained. These show that our approach scales gracefully as the number of components is increased. In fact by increasing the components by 400% (i.e., from 600 to 2400) the response time increases by about 80%. It is worth mentioning that there will be a number of components for which a single leader cannot guarantee acceptable response times so that a configuration with multiple leaders may be required.

[5]This has been evaluated using the RunTime class which provides methods that estimate memory usage.

#Koala	#Arms	#Crusher	#Manager	Response time (ms)
1	1	1	3	40
10	1	1	12	41
20	1	1	22	40
30	1	1	32	40
40	1	1	42	41
50	1	1	52	42
60	1	1	62	43

Figure 11. Experiment 1: response time (ms).

#Koala	#Arms	#Crusher	#Manager	Response time (ms)
600	1	1	12	521.3
1200	1	1	22	540.3
1800	1	1	32	691.1
2400	1	1	42	998.4

Figure 12. Experiment 2: response time (ms).

VII. Related work

Various languages and generation tools are available to specify and automatically generate control system implementations. In [10] the authors use an aspect-oriented approach in order to generate the global behaviour automatically. Our global state machines are a more structured way to specify the global behaviour and can be used in property verification. The authors in [11] propose a monitoring-oriented approach. System requirements are expressed using languages such as temporal logic and specifications are verified against the system's execution and user-defined actions can be triggered upon violation of the formal specifications. Although this approach allows the specification of global behaviour, it is verified by a centralised server. In contrast, in our approach all conditions and predicates are executed locally.

In the area of service-oriented computing, approaches such as ORBWork [12] provide orchestration of distributed

web services. None of these systems handle dynamic systems or failures nor do they automatically decompose orchestrations into choreographed execution.

In the area of self-adaptive systems, many works suppose that application components are distributed but few consider the ramifications of doing so. One exception is the work of Georgiadis *et al.* [13] in which every component has a manager and attempts to maintain architectural constraints (in response to node failures etc.) by acquiring a global change lock and applying structural repairs. However, this approach was not scalable, and only addressed architectural concerns.

Our approach is related to our previous work presented in [14], [15] where we automatically distribute a global state machine specification in order to build a distributed monitoring system. Although the authors distribute a global specification the distribution runs on a closed set of component instances. Every time a local state machine must move, it broadcasts a signal to all others to acquire the right to move the global state. Our approach reduces the amount of overhead since just one message is needed to acquire the permission. Moreover it introduces the notion of sets so that components can leave and join the system at runtime.

The GOANNA consensus protocol is based on Paxos [5], [8]. The Paxos protocol is used to solve the consensus problem between a set of nodes. However as described in Section V-B we have enhanced the protocol with further requests to implement action execution and introduced timeout events to ensure recovery in case of failure.

VIII. Conclusions

We have presented the GOANNA framework which permits logically-centralised adaptive behaviours to be decentralised in a robust, correct fashion. Global behaviours described using our state machine language can be decomposed into state machines local to each host of the distributed system and executed such that the local state machines implement the global state machine. A leader is used to guarantee correctness through a consensus protocol based on Multi-Paxos. The consensus protocol tolerates the failures of normal hosts and of the leader. Moreover, the approach is robust to changing availability of components, which are permitted to join and leave sets at runtime. Our experiments show that the approach is efficient in memory and is scalable to many components, particularly since we do not require that every component have its own manager.

Future directions include combining our system with abstract modelling tools such as LTSA [16] and PRISM [17] in order to provide a platform for both analysing and deploying distributed applications. There is also interesting work to be done on integrating the protocol fault handling with adaptive techniques. For example, if a certain component often fails, a higher-level decision procedure may choose to adapt by not using that component in future.

IX. Acknowledgements

This research was supported by UK EPSRC research grant EP/D076633/1 (UBIVAL). We gratefully acknowledge the feedback from our colleague Jeff Magee and also thank the members of the Policy Research Group and Distributed Software Engineering Section for their continued support.

References

[1] B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. de Lemos, "Software engineering for self-adaptive systems: A research road map," in *Software Engineering for Self-Adaptive Systems*, 2008.

[2] S.-W. Cheng, "Rainbow: Cost-effective software architecture-based self-adaptation," Ph.D. dissertation, 2008.

[3] N. Arshad, D. Heimbigner, and A. L. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems," *Software Quality Journal*, vol. 15, no. 3, pp. 265–281, 2007.

[4] L. Mostarda and N. Dulay, *GOANNA: State machine monitors for sensor systems*. www.doc.ic.ac.uk/˜lmostard/goanna, 2008.

[5] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[6] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From Goals to Components: A Combined Approach to Self-Management," *Proc. of ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2008.

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

[8] ——, "Paxos made simple, fast, and byzantine," in *OPODIS*, 2002, pp. 7–9.

[9] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer, "Assumption generation for software component verification," in *ASE*, 2002, pp. 3–12.

[10] F. Cao, B. R. Bryant, C. C. Burt, R. R. Raje, A. M. Olson, and M. Auguston, "A component assembly approach based on aspect-oriented generative domain modeling," *Electr. Notes Theor. Comput. Sci.*, vol. 114, pp. 119–136, 2005.

[11] F. Chen and G. Rosu, "Towards monitoring-oriented programming: A paradigm combining specification and implementation," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 2, 2003.

[12] S. Das, K. Kochut, J. Miller, A. Sheth, and D. Worah, "Orbwork: A reliable distributed corba-based workflow enactment system for meteor2," Tech. Rep., 1997.

[13] I. Georgiadis, J. Magee, and J. Kramer, "Self-organising software architectures for distributed systems," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM Press, 2002, pp. 33–38.

[14] P. Inverardi and L. Mostarda, "A distributed intrusion detection approach for secure software architecture," in *EWSA*, 2005, pp. 168–184.

[15] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili, "Synthesis of correct and distributed adaptors for component-based systems: an automatic approach," in *ASE*, 2005, pp. 405–409.

[16] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Ltsa-ws: a tool for model-based verification of web service compositions and choreography," in *ICSE*, 2006, pp. 771–774.

[17] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker, "Prism: A tool for automatic verification of probabilistic systems," in *TACAS*, 2006, pp. 441–444.