Distributed Fault Tolerant Controllers*

Leonardo Mostarda, Rudi Ball, Naranker Dulay
Department of Computing
Imperial College London
Email: {lmostard, rkb, nd}@imperial.ac.uk

Abstract

Distributed applications are often built from sets of distributed components that must be co-ordinated in order to achieve some global behaviour. The common approach is to use a centralised controller for co-ordination, or occasionally a set of distributed entities. Centralised co-ordination is simpler but introduces a single point of failure and poses problems of scalability. Distributed co-ordination offers greater scalability, reliability and applicability but is harder to reason about and requires more complex algorithms for synchronisation and consensus among components.

In this paper we present a system called GOANNA that from a state machine specification (FSM) of the global behaviour of interacting components can automatically generate a correct, scalable and fault tolerant distributed implementation. GOANNA can be used as a backend for different tools as well as an implementation platform in its own right.

1 Introduction

Programmers often face the problem of correctly co-ordinating distributed components in order to achieve a global behaviour. These problems include sense and react systems [9], military reconnaissance and rescue missions [1], autonomous control systems as found in aviation and safety critical systems.

The common approach used is to build of a centralised control system that enforces the global behaviour of the distributed components. The advantages of centralised co-ordination are that implementation is much simpler [11, 17] as there is no need to implement synchronisation and consensus among components, furthermore many tools are available for the definition and implementation of centralised controllers [2]. Existing distributed solutions are typically application-specific and require that the programmer understands and implements (often subtle) algorithms for synchronisation and consensus [7, 5].

*This research was supported by UK EPSRC research grant EP/D076633/1 (UBIVAL).

In this paper we present a novel approach to generate a *distributed* and *fault-tolerant* implementation from a single Finite State Machine (FSM) definition. We model the system as a set of components providing and requiring services. Co-ordination (global behaviour) is defined by a global FSM that defines the interactions among *sets* of components. Sets provide support to *group* available components at runtime and allows the selection of an alternative instance of a component in case of failure. A global state machine is automatically translated into a collection of local ones, one for each set. A FSM Manager at each host is responsible for handling the events and invocations for its local state machines and ensuring correct global behaviour. A Leader is responsible for the management and synchronisation of FSM Managers. This is achieved through an extension of a Paxos-based consensus protocol that implements correct, scalable and fault tolerant execution of global FSM. In particular scalability is obtained by using different optimisations that are derived from the FSM structure.

Various approaches could benefit from having automatically generated distributed implementations provided by a centralised specification. For instance the automatic synthesis of component based applications such as [4, 10, 19] are commonly used to generate a centralised controller where global state machines are obtained through composition and can have millions of states. Such approaches would benefit from our distribution approach that ensures correctness and provides scalability.

We have implemented our approach in a system called GOANNA [16] that takes as input, state machines and generates as output, distributed implementations in JAVA, C or nesC. The system is being used to develop distributed applications for sensor networks, unmanned vehicles and home networks [20, 18] and could be used as backend for tools that produce centralised controllers using finite state machines [19, 4].

The main contributions of this paper are the following:

1. The GOANNA platform that supports the co-ordination of components as a state machine specification and automatically generates a distributed imple-

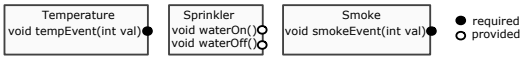

Figure 1. Fire Alarm Components

mentation utilising a Paxos-based consensus protocol.

2. We show that distributed co-ordination is equivalent to the central one.

3. We show that the system guarantees the global behaviour in the presence of node and communication failures or new nodes discovered at runtime.

4. We show that our optimisations significantly increase scalability with respect to the number of components and present the space and time overheads of the runtime system.

2 Overview

In this section we overview how we describe and distribute the co-ordination for a small fire alarm sensor system [20]. The system is composed of temperature and smoke sensors and a sprinkler actuator. The basic requirement is that the sprinkler should operate only when the temperature and smoke readings exceed a threshold. Defining the coordination requires that the programmer understands the underlying system model, the fault tolerance model and how to specify state machines using events, sets and signalling primitives in GOANNA.

2.1 System model and state machines

We assume the system is composed of a set of components that provide and require services. Components can be already bound together. In Figure 1 we show the component related to the fire alarm application. The component `Sprinkler` provides the services `waterOff()` and `waterOn()` to enable and disable water flow. The components `Temperature` and `Smoke` require the services `tempEvent(int val)` and `smokeEvent(int val)` where `val` denotes the value of the temperature and smoke, respectively.

Our global FSM specifies the sequence of events (resulting from component interactions) that are permitted in the running system and can proactively invoke service. More specifically, a global state machine is defined by a list of event-state-condition-action rules defined in terms of participating components (grouped into sets as described below). In Figure 2 we show the state machine related to our fire alarm application. Each rule states that when the system is in a given state, the related event is observed and the condition is true then an action is applied. For example the rule of line `10` states that when the state is `1`, the event `smokeEvent` is observed on a `smoke` and the smoke value is greater than `20` than the water must be enabled and the state changed to `2`.

```
1  global fsm fireAlarm(set Temperature temperatureSet,
2    set Smoke smokeSet,set Sprinkler sprinklerSet){
3
4    tempEvent  on  temperatureSet from *
5      0-1: event.val>50  -> {}
6      3-4: !(event.val>50) -> {signal to sprinklerSet
           waterOff();}
7
8    smokeEvent on smokeSet from *
9      1-0:  !(event.val>20) -> {}
10     1-2:  event.val>20 -> {signal to sprinklerSet
           waterOn();}
11
12   waterOn on sprinklerSet from *
13     2-3: {} -> {}
14
15   on timeout(10000)
16     2-2:{} -> {signal to sprinklerSet waterOn();}
17
18   waterOff on sprinklerSet  from *
19     4-0: {} -> {}
20 }
```

Figure 2. The GOANNA global FSM.

2.1.1 Events

Component interactions currently map to four GOANNA events, two for client endpoints (outgoing call and returned reply) and two for server endpoints (incoming call and outgoing reply). In the global finite state machine of Figure 2 we show events expressed using our syntax. For example, the event `"tempEvent on temperatureSet to *"` corresponds to a `tempEvent` service call from a `Temperature` component inside the temperatureSet. In this case we do not specify the receiver of the event (a hardware sensor).

Timeout events are also supported and are generated by GOANNA when no rule has been applied within the specified time t. For example, `timeout(10000)` will raise a timeout after 10 seconds if no other rule has been applied.

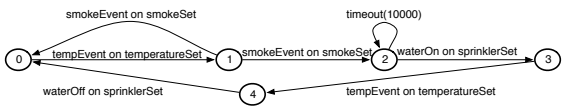

Figure 3. The graphical form of the global FSM

```
configuration fireAlarmConfiguration (floor:int)
  set t:Temperature where place == floor
  set sm:Smoke where place == floor
  set sp:Sprinkler where place == floor

  instance is:fireAlarm(t, sm, sp);
```

Figure 4. Fire alarm configuration

2.1.2 State-condition-action rules

For each event the global FSM can define a list of state-condition-action rules. A rule is of the form $q_s - q_d : condition \rightarrow action$ where q_s and q_d are states. When the event is observed, the state of the global FSM is q_s and the condition is true then the action can be applied and the state changed to q_d. When an event is observed but no rule can be applied (the condition does not hold or there are not relevant transitions) then a *reaction policy* can be applied such as *discard* (the event.)

2.2 System configuration

GOANNA system configurations specify both component sets and global FSM instances. Sets provide support to *classify* components as they are discovered and allows the selection of an alternative instance of a component in case of failure. Sets are grouped by component type and by a `where` predicate that can use attributes such as host name, position, node capabilities, to group components when they are discovered. In Figure 4, the three sets: `t`, `sm` and `sp` will group all components of types `Temperature`, `Smoke` and `Sprinkler` respectively running on floor `floor` of the building. Components can join and leave the set at run time. When an action from the global FSM must be performed an instance from the appropriate set is selected. This removes the need to manage the availability of components from the state machine specification and allows the selection of a new component in case of failures.

Sets are used to implement the following asynchronous best-effort primitives: (i) *signal to set call* and (ii) *signal to c in set call*. The former is used to invoke the method `call` on all components belonging to set while the latter to call the same service on exactly one component c.

Global FSM definitions can be multiply instantiated. For example, for our fireAlarm application we could instantiate a global FSM for each floor in a building.

2.3 Distribution

GOANNA automatically decomposes a global FSM into a collection of local ones (see Figure 5), one for each set plus a special FSM which contains all timeout events defined in the global FSM. In the following we refer to this

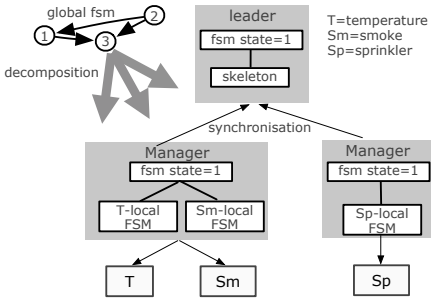

Figure 5. Centralised Control System - Distribution Implementation

timeout state machine as the skeleton. The FSM Manager local to each host uses the local FSMs while the leader uses the skeleton to implement the global FSM. More specifically the leader contains the correct state of the global FSM. When a FSM manager must validate an event it uses its local FSMs and its local state of the global FSM. This state (even if out of date) can be sufficient to reject locally the event reducing the number of synchronisations with the leader. If the FSM manager can accept the event it tries to *propose* a new state to the leader. The leader denies the propose when the FSM manager has an outdated state (in which case the leader updates the FSM manager with the correct state), otherwise the leader grants the propose. In this case the FSM manager performs the actions from its local FSM and synchronise leader with the new state.

2.4 State machine consensus protocol

Our consensus protocol extends Multi-Paxos with Steady State [14] with additional information in order to have a correct distributed state machine implementation. More specifically, it adds all information needed to execute actions (from the FSMs) and correctly parse system traces. This is achieved using timeouts to manage the one to one communications between FSM managers (executing the action) and the leader checking it. Multi-Paxos is normally described using client, acceptor, learner, and leader[1] roles. In our implementation the client, acceptor and learner roles are included in our FSM Manager.

The basic idea is that a FSM manager locally verifies event acceptance before proposing its new state. After a new state proposal the leader can either decline the request (e.g., the FSM 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 cor-

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

rect distribution they are not efficient in terms of memory and traffic overhead. State machines (automatically generated from high level tools [4, 19]) can be composed of millions of states so their deployment on each host can be inefficient. Moreover, FSM managers could continuously propose their new local states overloading the network. Our global state machine distribution process offers a partition of the FSM transitions and are loaded only when needed. Our protocol takes advantage of the state machine structure in order to avoid useless protocol instances. The idea is that an outdated local state can be enough to reject an event (see Section 3.6 for details).

2.5 Fault tolerance model

In GOANNA we make the following assumptions: (i) software components fail independently from their FSM Manager; (ii) FSM Managers can fail and recover; (iii) the leader fails and stops (but a new leader from a ranked set of nodes will be chosen) ; (iv) the ranked leaders control each other using reliable communication; (v) we assume a set of backups that are used as a stable storage for the last state accepted by the leader.

These assumptions are used to guarantee that a transition of a global FSM is performed if a FSM Manager can select an available component, the leader is running and the majority of backups are running.

3 Distributed state machine co-ordination

In this section we describe in detail how GOANNA generates a distributed state machine implementation from a global FSM specification.

3.1 The system model

We first introduce some notation used to describe the system model. The set E denotes the set of all possible component events while $e_1, \ldots e_n$ are elements in E. The set E^c denotes the set of events locally observed on a component c and $e_1^c \ldots e_n^c$ elements in E^c. We use T_s to denote all possible traces (i.e., sequence of events) inside the system. We use T_c to denote all traces local to a component c.

Traces are subject to the *happened-before* relation ($\rightarrow$) [13], i.e., a message can be received only after it has been sent. In the following we define the merge of component traces.The basic idea is that the merge of two or more traces should conform to the following two rules: (i) all independent events from different traces can occur in any order in the merged trace; (ii) events within the same trace must retain their order. A trace resulting from this merge is usually called a *linearisation* [3].

Definition 1 *Let $S = \{c_1, \ldots, c_i, \ldots\}$ be a system. Let T_{c_1} and T_{c_2} be the local traces of the component instances $c1$ and $c2$, respectively. A* merge trace *$T_{c_1} \oplus T_{c_2}$ is a new trace defined by $e_1\ e_2 e_3 \ldots e_t \ldots$ where: (i) e_t appears in $T_{c_1} \oplus T_{c_2}$ if and only if e_t appears either in T_{c_1} or T_{c_2}; (ii) for each e_i and e_j, with $i < j$, $e_i \rightarrow e_j$.*

In order to prove the correctness of our distributed implementation we need to prove that the merge of all FSM manager traces is accepted by the global state machine (see appendix A).

3.2 State machine definitions

In the following we provide definitions for state machines and the related acceptance criterion.

Definition 2 *A state machine is a 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.*

Event outside the FSM alphabet are ignored (i.e., they are not subject to the FSM validation). When different state machines are defined the event must be accepted by all of them. We emphasis that T_A is a subset of T_s (all possible system traces). More specifically a global FSM define all permitted traces inside the system.

3.3 Local state machine generation

In order to distribute the global state machine, we automatically decompose it into a set of local ones, one for each set, and a skeleton. We will use $A = (Q, q_0, I, rules)$ to denote a global state machine, s^c to denote a set s defined over the component type c and $A_{s^c} = (Q_{s^c}, q_{s0}, I_{s^c}, rules_{s^c})$ to denote the local state machine assigned to the set s^c.

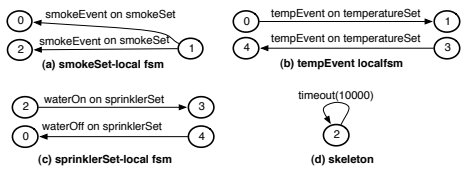

Figure 6. Generated local state machines and skeleton

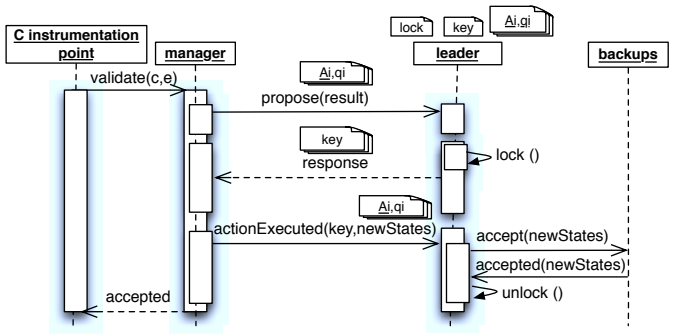

Figure 7. Successful protocol execution

In order to generate all local state machines we consider all sets defined in the global FSM. For each set s^c we generate the local state machine A_{s^c} by examining the global state machine A for rules of the form $R = (e^c, q_s, q_d, condition, action)$. 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 of the type c belonging to the set s^c. The skeleton A_k contains the list of all time out rules.

In Figure 6 we show all local state machines generated from the global state machine of Figure 3. We emphasise that each transition has been projected locally to the set that its event relates to. Effectively our distribution algorithm defines a partition of the global state machine transitions.

3.4 Successful protocol execution

In Figure 7 we show the global flow of a successful protocol execution. We denote with $\underline{Ai}$ a component instance of the type A. The protocol starts when an instrumentation point related to a component instance c detects an incoming/outgoing message. This generates an event e and invokes the procedure `validate(c,e)` on its local FSM manager. This uses its local state q_i of $\underline{Ai}$ to accept the event. More specifically the FSM Manager finds all set s^c the component c belong to, loads the related local FSM A_{s^c} and uses qi to accept (see Definition 3 for the definition of acceptance) the event e. If the event can be accepted the FSM manager starts the protocol by sending a `propose(result)` request to the leader containing the fsm instance name $\underline{Ai}$ and new proposed state q_i. The leader receives the request and compares the received state q_i with its local state, e.g., $\underline{qi}$. Moreover it checks whether or not the fsm $\underline{Ai}$ has been locked by another FSM manager. Suppose that the states are the same ($q_i == \underline{qi}$) and no fsm instance has been locked. Then the leader generates a new key and responds with a `response` data structure to the FSM manager. This structure contains a `key` (denoting the protocol instance) and an `outcome` (set to `accepted`). With this answer the leader promises to the FSM manager the lock on the required fsm instance $\underline{Ai}$. The FSM manager receives the response, performs the local actions (from the rules of the local FSM), and sends back to the leader an $actionExecuted(key, newStates)$ response where $newStates$ contains the new state after the execution of the rule. The leader receives the request and checks the existence of the key. In case the key exists it deletes the key, unlocks the fsm instance $\underline{Ai}$ and updates its local state with the received one. The process of updating the state requires to perform a Multi-Paxos protocol with Steady State. More specifically, the new state is sent to a set of backups through an `accept` request. When the majority of them notify the update (through an `accepted` request) the protocol can correctly terminate.

When multiple state machine instances are defined the FSM manager must check the event acceptance for all of them. As for the aforementioned execution if the event is accepted the FSM 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).

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

3.5 Protocol exceptions

A protocol instance can raise a *manager out-of-sync* and fsm instance *locked* exceptions. A *manager out-of-sync* exception is raised when any of the state sent by the FSM manager and the leader one are different. This is a consequence of a FSM 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 an out-of-sync error containing its state (i.e., the most updated one). This is used

by the FSM to updated its local state. A *locked* exception is generated when a FSM manager proposes a state related to an instance $\underline{Ai}$ that has been locked by another FSM manager. In this case the leader sends back a `response` data structure with the *locked* error.

Failures on FSM 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 FSM managers.

A leader can see a FSM 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 FSM manager fault, a communication failure or a slow (overloaded) FSM manager.

- A *propose response failure* occurs when the leader fails to communicate to a FSM manager the outcome of a proposal of states (i.e, a `response` data structure). In this case a time out is raised and the leader deletes any key or lock granted.

- An *action execution timeout* occurs when a FSM 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 FSMs to be unlocked. It is worth mentioning that even if the FSM manager sends an $actionExecuted$ invocation after the timeout expires this will be detected (the key is no longer existent) and the FSM states 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 FSMs (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 FSM manager but fails to acknowledge the reception. In this case the leader ends the protocol instance and waits for the next request.

A FSM 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 (*actionExecution 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. In all cases the FSM Manager ends the protocol execution and returns an error to the instrumentation point. We emphasis that for actionExecution invocation and actionExecution response failures there is the possibility that the FSM Manager performs invocations from its local FSMs. In the case of non-recoverable actions that the global execution can be inconsistent.

In our protocol, we have a set of ranked leaders. While the highest ranked leader is servicing FSM Managers the lower ranked leaders monitor the highest ranked leader for failure. More specifically when the highest ranked leader is no longer detected, the next leader in the rank is elected. This recovers all correct global states from the backups.

An error on the protocol execution is always returned to the instrumentation point that can be programmed to implement different reactions such as *retry* the parsing, *discard* the event and so on.

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 FSM manager is always granted permission and always fails. In order to avoid this kind of livelock the leader always chooses a random FSM manager when granting permission.

3.6 Protocol optimisations

In this section we describe various optimisations that are based on the structure of the state machine. In other words while our protocol solves the general problem of consensus among FSM managers, the state machine can guide us in avoiding useless protocol instances. In particular we have implemented the following optimisations: *drop duplicate requests*, *grouping* and *drop unreachable requests*

In *drop duplicate requests* the FSM manager buffers each $result$ data structure that has been sent with a `propose` request. Any further propose that contains the same state machine instance $\underline{Ai}$ with the same state qi is locally buffered and held until the first request has returned its result. If the result contains an error related to $\underline{Ai}$ then the same error is returned for all instrumentation points, otherwise if the request has been accepted the FSM manager waits for the action to complete and releases one of the requests.

Grouping allows different operations to be sent in the same message reducing the amount of messages sent. For instance all $signal$ requests related to the same action execution are grouped together and sent in a single message.

The *drop unreachable requests* optimisation avoids

sending $propose$ requests that are certain to be dropped. This is based on the $reachable_A : Q \times Q \rightarrow Bool$ function that is derived from the structure of a global state machine A. In particular, $reachable_A(q_s, q_d)$ is true when the state q_d is reachable from the state q_s and false otherwise. The FSM manager keeps track, for each instance $\underline{Ai}$, of the last updated state q_s. Before proposing a new state q_d the FSM manager verifies $reachable(q_s, q_d)$. When $reachable(q_s, q_d)$ is false the event is locally rejected without interacting with the leader. Effectively, the proposed state q_d cannot be reached from $\underline{q_s}$.

4 Evaluation and Results

GOANNA for Java 1.5 was evaluated on a 100 Mbit network using a cluster of 50 Intel Pentium architecture machines each operating with at least 2 GB of RAM running the Linux operating system. A single Leader and as many as 2600 Components (sensors) were executed.

Experiments sought to (a) validate the GOANNA implementation, (b) measure the outcome of induced faults (killing and rebooting hosts) and (d) highlight the performance optimisations resulting from using the FSM structure.

We first determined the memory overhead, protocol overhead and baseline *Average Event Time* (*AET*) per FSM Manager. AET represents the time taken for the *validate(c,e)* (Figure 7). In effect it measures the time it takes for a FSM Manager to *validate* a component interaction event. We also consider this the average number of requests that a Leader can handle per second (*throughput*). Effectively the latter measures the additional traffic generated by our distributed state machine implementation.

All experiments created FSM Managers and allocated a set of Components (sensors) to each. The scenario used considered a GOANNA hierarchy made up of a single Leader, multiple FSM Managers and multiple Components. A typical system consisted of a Leader (L), multiple FSM Managers (M) and bound Components (C), where arrows represent directional send-receive communications (Figure 8). We assumed various sensors (Components) to bind to FSM Managers at start-up - hence no allocation algorithm was applied during runtime, rather sensors were statically bound to FSM Managers.

4.1 Memory Overhead

A FSM Manager performs three main functions: (i) checking conditions; (ii) executing the consensus protocol and (iii) executing state machine actions. Functions (i) and (iii) can be arbitrary code, but are typically simple boolean expressions or calls to methods/services. Successful execution of the consensus protocol requires four message exchanges between a FSM Manager and the Leader.

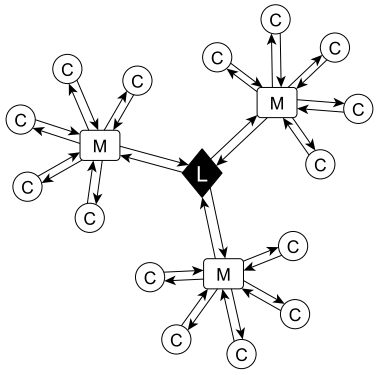

Figure 8. Example GOANNA Topology

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

Table 1. Protocol overhead

The overheads of the exchanged messages are summarised in table 1. We note however, that actions can generate additional traffic through *signal* calls. These are performed locally by the FSM Manager and their cost is equivalent to an inter-host remote procedure calls in the implementation language (e.g. the sending of the parameters and the service identifier). A major cost of this is the processing of validate requests, which, if not distributed among several FSM Managers, will lead to a bottleneck equivalent of a centralised solution.

Table 2 summarises the memory costs of the FSM Manager and leader for systems with 10, 80 and 120 components. In the worst case with 120 components running, the FSM Manager and leader memory (both heap and data) is 489KB and 902KB, respectively.

Table 3 shows the sizes of the Global FSM and each of the generated local FSM machines for the Fire Alarm ex-

Process	Components	Heap (KB)	JVM (MB)
leader	10	366	10
leader	80	464	11
leader	120	489	13
FSM manager	10	697	11
FSM manager	80	763	13
FSM manager	120	902	14

Table 2. Leader and FSM Manager memory consumption.

FSM	Size (KB)
global fsm	3
Temperature	1.5
Smoke	1.5
Sprinkler	1.4

Table 3. State machine file size

ample. The sizes correspond to sizes of the serialised object for each FSM.

4.2 Execution Overhead

Execution overhead was measured using AET to maintain consistency between experiments. The evaluation was performed on several configurations to show the effect of increasing distribution: (A) one FSM Manager and three Leaders; (B) two FSM Managers and three Leaders; (C) three FSM Managers and three Leaders; (D) ten FSM Managers and three Leaders.

We ran systems with between 10 to 125 smoke and temperature sensor components. Each sensor Component was run in a separate thread and sent a reading every 400 ms. For configuration A, all sensors were run on the same host. For configuration B, two hosts ran half of the sensor instances each. For configuration C, a third of the sensors were run on each host. For configuration D, a tenth of the sensors were run on each host.

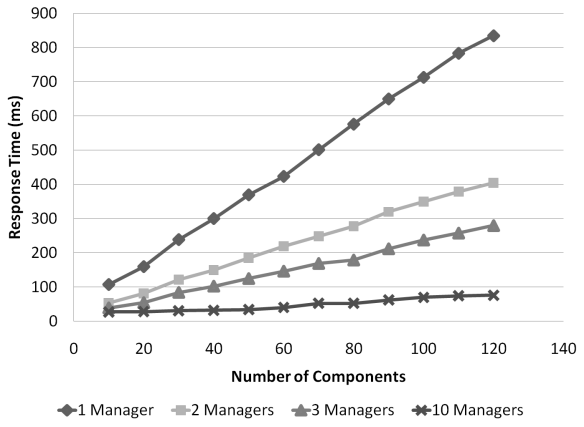

Figure 9. Event Validation Time

In Figure 9 we show a FSM Manager's response time for all four configurations. Each value was obtained by running the experiments for 10 minutes and calculating the average of all validate request response times experienced by every sensor. For example, in the case where 125 sensors were running at the same time, the results were as follows. In configuration A, the average time for a FSM manager to validate a component interaction was 834 ms. In the configuration B, it was 404 ms. In configuration C, it was 280 ms. In configuration D, it was 76 ms.

This shows that the protocol scales linearly when components are distributed across different hosts. The critical bottleneck is the FSM manager and not the leader. The FSM manager performs most of the computation, i.e., creates a new thread for each component instance, verifies acceptance and applies the actions. The leader only responds to requests by sending a few integers. The more FSM managers (hosts) we have, the more efficient the implementation is.

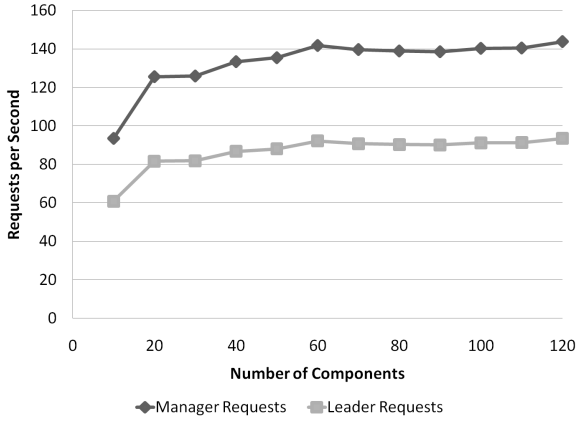

Figure 10. Throughput of validation requests

Figure 10 shows the throughput of the Leader and FSM Managers, that is the number of validate requests handled per second. For a single FSM Manager and a Leader, the FSM Manager receives an average of 140 requests per second while the leader receives 83 (i.e. 35% less), a significant improvement to reducing processing and communication load. It is this difference in performance between FSM Manager Requests and Leader Requests which is noteworthy. This improvement is a consequence, both the scalability of the architecture and our optimisation process, since most of the requests are dropped because they cannot be accepted in the current state of the FSM Manager.

4.3 Average Event Time Performance Measurement

We considered a standard system containing a three Leaders, m many FSM Managers and 52 Components per FSM Manager (50 temperature sensors, 1 smoke sensor and 1 sprinkler). Components provided stress-test sensor readings as common with the previous benchmark experiments (every 400 ms). Performance was measured according to a standard instantiation of the GOANNA system: (1) three Leaders were created on separte hosts within the cluster; (2)m many FSM Managers were created; (3) $m \times c$ many

Components were created and allocated equally between FSM Managers (distribution), where c represented the number of Components allocated to a single FSM Manager.

An increase in FSM Managers multiplied the number of Components providing sensor data to the Leader, using the hierarchical communication network we saw a distribution of load commonly seen in hierarchical network architectures (Figure 11). This would place increased load on the Leader. GOANNA was considered to be effective if it could limit this Component load.

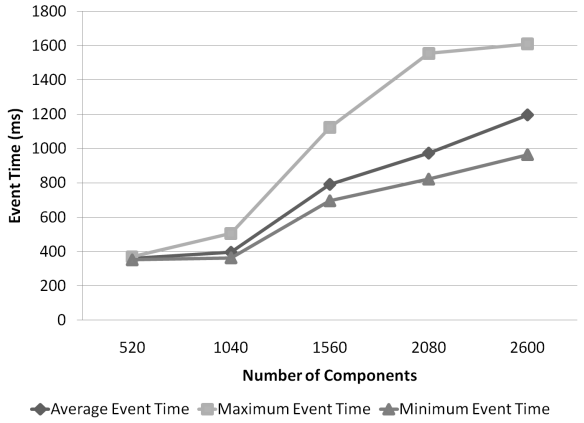

Figure 11. Base Average Event Time Performance

Indeed, baseline performance was seen to increase from 360 ms for 520 Components to 1196 ms for 2600 Components a change of 835 ms with a 5 fold increase in total Components - 167 ms cost per 520 Components added to the system.

This performance reduction while significant presents the opportunity to further distribute load amongst Leaders for the maintenance of a low AET. In other words, users of GOANNA can determine according to AET the optimal number of Components, FSM Managers and Leaders to use to achieve a specific AET performance, based on the architecture a base line benchmark is made. GOANNA is scalable and this scalability is both measurable and predictable.

4.4 Fault Tolerance

While the previous performance tests had considered GOANNA in simple initialisation and computation phases, faults were introduced to measure the capacity for GOANNA to deal with the addition (booting) and removal (killing) of FSM Managers and Components at runtime. Two experiments were conducted in relation to the base scenario, namely (a) *kill* and (b) *kill and reboot*. Two phases occurred in each experiment: Initialisation and Fault Injection, where Fault Injection either killed or rebooted FSM Managers in sets of 10, 20, 30, 40 and 50 FSM Managers.

Three Leaders (L) and a collection of m FSM Managers (M) were started and components allocated. Each FSM Manager's components were instructed to operate and provide sensor readings for 30 seconds.

4.4.1 Kill

In each experiment a set of 50 FSM Managers were seen to operate normally. We introduced faults by killing the Management processes on individual hosts. Given the kill scenario, once a FSM Manager was removed from the GOANNA system it was *not* restarted. A reduced load of FSM Managers on the Leader was seen to improve the performance of GOANNA, with reduced AET existing after a FSM Manager had been killed, however the loss of a FSM Manager lost subset of Components providing data to the system. Figures 12 and 14 represents the final AET value reported for the Kill experiment, illustrating the reduced AET where increasing the number of FSM Managers killed. Significant maximum values exist due to GOANNA timeouts occurring after FSM Managers have been killed. Figure 12 illustrates the Maximum, Minimum and Average Event Time occurring where FSM Managers are removed from the GOANNA system. Where FSM Managers are not rebooted, we see improved overall performance of GOANNA - fewer FSM Managers present a reduced load on the Leader. If all 50 FSM Managers are killed, there is no system provision.

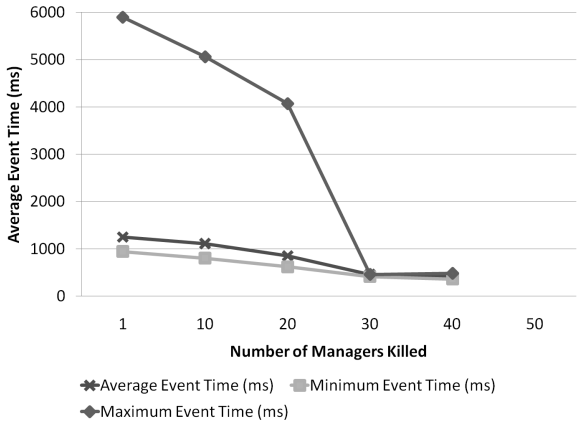

Figure 12. Kill Performance

The Leader was seen to handle the removal and time-out of Management messages sufficiently not to let the entire system be affected by the loss of specific FSM Managers. In this respect GOANNA proved successful at maintaining operation given these changes, providing a mean performance per FSM Manager which was acceptable to the continued

operation of the GOANNA system.

4.4.2 Kill-Reboot

As an derivative test, GOANNA was seen to perform consistently and normally where FSM Managers were first killed and then rebooted. As rebooting restarted Management processes the final effect was seen to not affect the AET of the overall system. In contrast to the results produced by the removal of FSM Managers, AET has limited fluctuation between 1143 ms and 1271 ms for increasing FSM Managers. Figure 13 shows the Maximum, Minimum and Average Event Time occurring where FSM Managers are killed and then rebooted.

Figure 13. Kill-Reboot Performance

The process of rebooting FSM Managers restores the GOANNA system to normal processing and hence normal AET results occur for a capacity benchmarked system.

4.4.3 Comparison

GOANNA is seen to continue to execute normally where FSM Managers were removed from the system. A Leaders performance was seen to adapt to the loss of FSM Managers, improving provision of service to FSM Managers which still existed. We see this as a reduction in AET. Killing up to 40 FSM Managers resulted in a reduced AET of 413ms (Figure 14).

In the instance where FSM Managers were rebooted the system performance was seen to behave as if no fault had occurred in the system - results mimicked a faultless system. GOANNA is hence capable of adapting to loads on a specific Leader, limited only by the capacity of the Leader to handle further FSM Managers. We can expect performance to be influenced minimally by the removal and addition of FSM Managers, based on the limitations of the network GOANNA is being executed on.

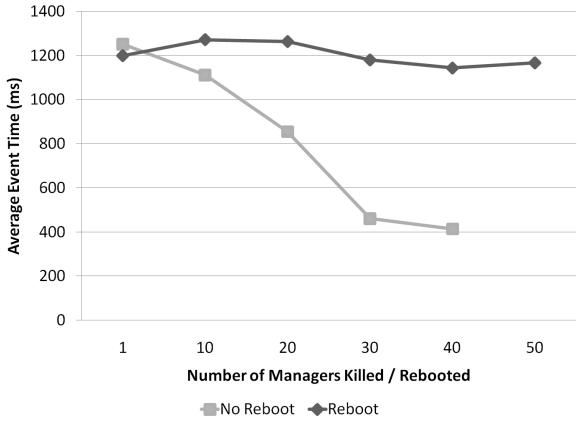

Figure 14. Comparison of Performance: Kill and Kill-Reboot

4.5 Summary

We should note that multiple factors affect the capacity of GOANNA to operate effectively, including network and host performance. This aside, performance data presents a scalable system where FSM Managers can be added and removed at runtime with predictable effect on the GOANNA hierarchy. We attribute the success of GOANNA to both the GOANNA protocols and the hybrid FSM approach used.

The increased AET outcome, given increased Components, commonly affects such distributed systems and was expected, however the performance degradation for systems exceeding 2600 components is a gradual change - we observe a shallow gradient (or derivative) in terms of AET performance loss (Figure 11). As such, it is possible to facilitate extra distribution of Leaders in a scalable and strategic manner, tuning the number of Leaders or FSM Managers to achieve a specific AET performance that a system builder deems acceptable for a given deployment of sensors. Furthermore, GOANNA is capable of tolerating node addition and subtraction gracefully. Faults occurring are contained and localised to specific FSM Managers.

The implementation of GOANNA considers the provision of three redundant Leaders. Where Leaders are killed the upper-bound performance of the system was limited by Leader response time which was depending on the underlining TCP timeout.

5 Related work

Various techniques have been developed in order to have a distributed implementation from a logically centralised specification. In [6] the authors use an aspect-oriented approach in order to automatically generate the global be-

haviour. They specify component definitions and aspects related to functional and non-functional requirements. Some of the aspects are used to weave components together. Our global state machines offer a more structured way to specify the global behaviour and can be used in property verification. In [8] the authors propose a monitoring-oriented approach. They combine formal specifications with implementation to check conformance of an implementation at runtime. System requirements can be expressed using languages such as temporal logic. Specifications are verified against the system 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. Our earlier work [12] performs state-machine monitoring, but on closed distributed systems and assumes no failures. GOANNA supports active co-ordination, dynamic systems, and fault-management using consensus. In [15] the authors present a workflow engine that migrates the workflow instance (specification plus run-time data) between execution nodes. In [21] they split the specification into several parts in order to have a distributed execution. However, this approach defines a set of independent communicating entities rather than a global behaviour.

6 Conclusions

In this paper we have described GOANNA, a system that models the co-ordination of component-based systems as a global state machine specification and automatically generates a correct, scalable and fault-tolerant implementation. GOANNA decomposes global state machines into local ones, and uses a consensus protocol to synchronise them. The system guarantees the global behaviour in the presence of failures and supports the introduction of new component instances at runtime. Performance for the GOANNA-Java version shows that distributed co-ordination scales with respect to the number of FSM managers (nodes).

References

[1] E. Asmare, A. Gopalan, M. Sloman, N. Dulay, and E. C. Lupu. A mission management framework for unmanned autonomous vehicles. In *MOBILWARE*, pages 222–235, 2009.

[2] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transaction on Dependable- and Secure Computing*, 1:11–33, 2004.

[3] M. Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd Edition)*. Addison-Wesley, 2006.

[4] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/SIGSOFT FSE*, pages 141–150, 2009.

[5] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[6] 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.*, 114:119–136, 2005.

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07*, pages 398–407, 2007.

[8] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[9] A. Deshpande, C. Guestrin, and S. Madden. Resource-aware wireless sensor-actuator networks. In *IEEE Data Engineering*, 2005.

[10] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, pages 3–12, 2002.

[11] R. Guerraoui and L. Rodrigues. *Reliable Distributed Programming*. Springer, 2006.

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

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[14] L. Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.

[15] F. Montagut and R. Molva. Enabling pervasive execution of workflows. *2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[16] L. Mostarda and N. Dulay. *GOANNA*. www.doc.ic.ac.uk/˜lmostard/goanna, 2008.

[17] P. Oppenheimer. *Top-down network design*. Cisco system inc., 2004.

[18] D. Pediaditakis, L. Mostarda, C. Dong, and N. Dulay. Policies for Self Tuning Home Networks. In *IEEE POLICIES 09*, July 2009.

[19] G. D. Penna, D. Magazzeni, B. Intrigila, I. Melatti, and E. Tronci. Automatic generation of optimal controllers through model checking techniques. In *ICINCO-ICSO*, pages 26–33, 2006.

[20] G. Russello, L. Mostarda, and N. Dulay. Escape: A component-based policy framework for sense and react applications. In *CBSE*, pages 212–229, 2008.

[21] R. Sen, G.-C. Roman, and C. Gill. CiAN: A Workflow Engine for MANETs. *Coordination Models and Languages*, pages 280–295, 2008.

A Correctness

We now outline a proof that our distributed implementation correctly executes global state machines. We assume that we have exactly one state machine instance $\underline{A}$ for a state machine definition $A = (Q, q_0, I, rules)$.

Definition 6 *Let $A = (Q, q_0, I, rules)$ be a state machine and let q be a state of A. A FSM manager M belongs to the set $managers(q)$ if there exists a rule (e, q, q_d, condition, action) defined in a local state machine A_{set}.*

In other words the set $managers(q)$ includes every FSM manager M such that a local state machine A_{set} contains a q-exiting rule. These are the only FSM managers that can perform a successful synchronisation when the leader's FSM instance is in state q.

Theorem 1 *Let $\underline{A}$ be a state machine instance and $A = (Q, q_0, I, rules)$ be its definition. Let $S = \{c_1, \ldots, c_n, \ldots\}$ be the component instances and $M = \{M_{c_1}, \ldots, M_{c_n}, \ldots\}$ the corresponding FSM managers. Let L be the leader. The state machine instance $\underline{A}$ accepts a system trace $Ts = T_{c_1} \oplus \ldots \oplus T_{c_n} \ldots$ if and only if all each traces T_{c_j} are accepted by each corresponding FSM manager M_{c_j} with $1 \leq j \leq n$.*

Each time a leader receives a message `actionExecuted` a FSM manager $M_{c_{i-1}}$ locally applies a rule (e_{i-1}, q_{i-1}, q_i, condition, action). This is consequence of a `validate(`c_{i-1}`,`e_{i-1}`)` request from component c_{i-1}. If we prove that the sequence or rules (e_{i-1}, q_{i-1}, q_i, condition, action), with $0 \leq i \leq n$, produces a trace $e_0, \ldots, e_n$ accepted by the global state machine A then we have proved that the merge of traces generated by the component instances is accepted by A. The proof is performed by induction on the length of the trace $e_0, \ldots, e_n$.

At the beginning, the leader's fsm instance is in state q_0 and each state machine instance $\underline{A}$, local to the FSM manager M_C, is in a state q_c. Our distribution scheme ensures that each rule (e, q_0, qd, condition, action) is projected on FSM managers belonging to the set $managers(q_0)$. Therefore only a FSM manager in this set can propose the state q_0 that is consistent with the leader one. Suppose that M_{C_0} in $managers(q_0)$ receives the request `validate(`C_0`,`e_0`)` from a component C_0. It first tries to perform the `tryAccept(`C_0`,`e_0`)` procedure to accept the event e_0. If the procedure finds a rule (e_0, qs, qd, condition, action), with $qs \neq q_0$, that can be applied. Then the new state q_s is proposed to the leader. The leader refuses the proposal (since it is in state q_0) and sends the correct global state q_0 back to the manger. The FSM manager M_{C_0} receives the state and performs the `update` function. Since M_{C_0} is in $managers(q_0)$ it contains at least a local state machine A_{set} where a transition of the form (e_0, q_0, qd, condition, action) has been projected. Therefore the update function returns the value `true`. In this case the state of $\underline{A}$ (local to the FSM manager M_{C_0}) is set to q_0 and the acceptance of the event e_0 is tried in the state q_0 (this is performed through the call `acceptEvaluation`). Suppose that the rule R=(e_0, q_0, $q1$, condition, action) can be applied. Then M_{C_0} proposes the new state q_0 to the leader. This receives the proposal and accepts it. The FSM manager can apply the rule R (that generates the trace e_0) and set the new leader state to $q1$. Since the rule R is a projection of the state machine definition A the trace e_0 is accepted by A.

Let $e_0, \ldots, e_n$ be the trace generated applying the rules (e_{i-1}, q_{i-1}, qi, $condition_i$, $action_i$), with $0 \leq i \leq n$. By induction we assume that the trace is accepted by $\underline{A}$ and we prove the acceptance of $e_0, \ldots, e_n, e_{n+1}$. We can prove that after the rule application the leader is in state qn (this would contradict the inductive hypothesis). Therefore only a FSM manager in $managers(qn)$ can apply a local rule (e_n, q_n, qd, $condition_{n+1}$, $action_{n+1}$). As for the base case a FSM manager $M_{C_{n+1}}$ can apply a rule (e_n, q_n, q_{n+1}, $condition_{n+1}$, $action_{n+1}$) and propose the state q_n. The leader will accept the propose and the FSM manager will apply the rule generating the new trace $e_0, \ldots, e_n, e_{n+1}$ that is accepted by A.

If we let $T = T_{s_1} \oplus \ldots \oplus T_{s_p}$ be a trace accepted by $\underline{A}$. We can always find a schedule (of events delivered to each FSM manager) such that each component c_i produces the trace T_{s_i}.

The proof can be easily extended to the case where multiple state machine instances are defined.