Distributed Fault Tolerant Controllers

Leonardo Mostarda, Rudi Ball, Naranker Dulay

Distributed Fault Tolerant Controllers*

Leonardo Mostarda, Rudi Ball, Naranker Dulay

Imperial College London,UK SW7 2AZ
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.

1 Introduction

Programmers often face the problem of correctly co-ordinating distributed components in order to achieve a global behaviour. These systems include sense and react systems, military reconnaissance and rescue missions, 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 components. The advantages of centralised co-ordination are that implementation is simpler [1, 2] 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 [3]. Existing distributed solutions are typically application-specific and require that the programmer understands and implements (often subtle) algorithms for synchronisation and consensus [4, 5].

In this paper we present a novel approach to generate a *distributed* and *fault-tolerant* implementation from a single Finite State Machine (FSM) definition of a global behaviour. 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

* This research was supported by UK EPSRC research grants EP/D076633/1 (UBIVAL) and EP/E025188/1 (AEDUS2).

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 [6, 7] 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 since it ensures correctness and provides scalability.

We have implemented our approach in a system called GOANNA 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 [8, 9] and could be used as backend for tools that produce centralised controllers using finite state machines [7].

The main contributions of this paper are: (1) the GOANNA platform that supports the co-ordination of components as a state machine specification and automatically generates a distributed implementation utilising a Paxos-based consensus protocol; (2) we show that the system guarantees the global behaviour in the presence of node and communication failures or new nodes discovered at runtime; (3) we show that our optimisations significantly increase scalability with respect to the number of components and present the time overhead of the runtime system.

2 Overview

In this section we overview how we specify and distribute the co-ordination for a small fire alarm sensor system [8]. 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.

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 our fire alarm system we have `Sprinkler` components providing the services `waterOff()` and `waterOn()` to enable and disable water flow and `Temperature` and `Smoke` components requiring 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

services. 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 1 we show the state machine for the 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 Smoke smokeSet, set
       Temperature temperatureSet,set Sprinkler
       sprinklerSet){
2
3    tempEvent  on  temperatureSet from *
4      0-1: event.val>50  -> {}
5      3-4: !(event.val>50) ->
6        {signal to sprinklerSet waterOff();}
7
8    smokeEvent on smokeSet from *
9      1-0:  !(event.val>20) -> {}
10     1-2:  event.val>20 ->
11       {signal to sprinklerSet waterOn();}
12
13   waterOn on sprinklerSet from *
14     2-3: {} -> {}
15
16   on timeout(10000)
17     2-2:{} ->
18       {signal to sprinklerSet waterOn();}
19
20   waterOff on sprinklerSet  from *
21     4-0: {} -> {}
22 }
```

Fig. 1. The GOANNA global FSM.

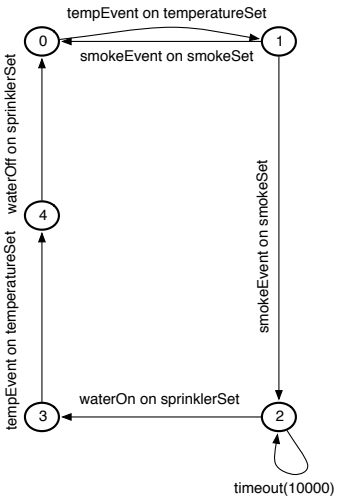

Fig. 2. The graphical form of the global FSM

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 1 we show events expressed using our syntax. For example, the event `"tempEvent on temperatureSet from *"` corresponds to a `tempEvent` returned reply on a `Temperature` component inside the temperatureSet. In this case we do not specify the component sending the reply (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.

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

```
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);
```

Fig. 3. Fire alarm 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 3, 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 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 fire-Alarm application we could instantiate a global FSM for each floor in a building.

2.3 Distribution

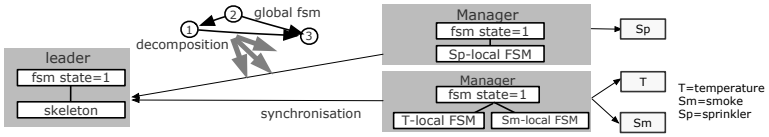

Fig. 4. Centralised Control System - Distribution Implementation

GOANNA automatically decomposes a global FSM into a collection of local ones (see Figure 4), 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 timeout state machine as the skeleton. The FSM Manager local to each host uses the local FSMs to validate the component interactions related to the host it resides on while the

leader uses the skeleton to execute timeouts. The leader also contains the correct state of the global FSM. When a FSM manager must validate an event (w.r.t. the behaviour defined by the global FSM) 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 proposal request 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 proposal request. In this case the FSM manager performs the actions from its local FSM and synchronises the leader with the new state.

2.4 State machine consensus protocol

Our consensus protocol extends Multi-Paxos with Steady State [10] 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 request 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 correct distribution they are not efficient in terms of memory and traffic overhead. State machines (automatically generated from high level tools [7]) 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 while 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

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

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 ($\rightarrow$) [11], i.e., a message can be received only after it has been sent. A system trace T_s can be obtained through a *linearisation* [12]. The basic idea is that all component-local traces can be merged by using 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.

3.2 State machine definitions

A state machine is used to define the correct traces in our distributed system. In the following we provide the FSM formal definition.

Definition 1. *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 2. *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 3. *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.*

The language T_A recognised by a state machine A is composed of all traces accepted by it. Events 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 defines all permitted traces inside the system.

3.3 Local state machine generation

In order to distribute the global state machine, we first need to decompose it into a set of local ones, one for each set, and a skeleton. We 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.

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 timeout rules.

Fig. 5. Generated local state machines and skeleton

In Figure 5 we show all local state machines generated from the global state machine of Figure 2. 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

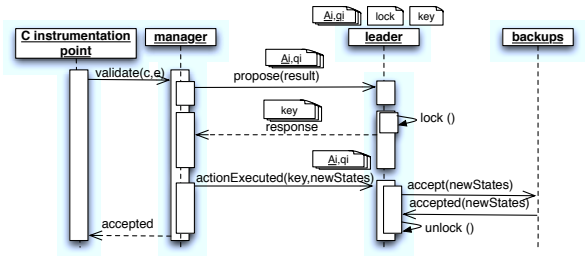

Fig. 6. Successful protocol execution

In Figure 6 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 finds all set s^c the component c belongs to, loads the related local FSM A_{s^c} and looks for a state qi where the event e can be accepted (see Definition 2 for the definition of acceptance). 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 uses 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).

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 their communication links are handled in our protocol using timeouts. They are needed to manage the asynchronous one-to-one communications between FSM managers (executing the action) and the leader checking it. 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 failures 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 timeout is raised on the leader side that 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 failures. 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 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 failures 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.

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.

Our distributed FSM implementation has been proved to be *correct* by showing that a linearisation of the traces produced by the FSM Managers (see Section 3.1) is always accepted by the global FSM (see the extended technical report of this paper for details [13]).

3.6 Protocol optimisations

While our protocol solves the general problem of consensus among FSM managers, the state machine structure can allow the addition of different optimisations.

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 number 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

A more extensive evaluation of results can be found in [13]. 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. As many as 2600 Components (sensors) were executed. Experiments sought to (a) validate the GOANNA implementation, (b) measure the outcome of induced failures (killing and rebooting hosts) and (c) highlight the performance optimisations resulting from using the FSM structure. The *Average Event Time* (*AET*) represents the time taken for the *validate(c,e)* to complete (Figure 6). AET measures the time taken for a FSM Manager to *validate* a component interaction event. *Throughput* was represented as the average number of requests that a Leader could handle per second.

Each experiment 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.

Execution Overhead Execution overhead was measured using AET to maintain consistency between experiments. The evaluation was performed on several configurations.

Experiments considered systems with between 10 and 125 components. Each sensor Component was executed 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 (summary in Figure 7).

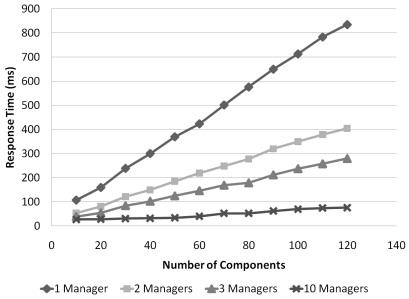

Fig. 7. Event Validation Time

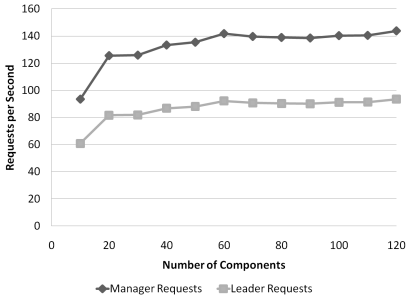

Fig. 8. Throughput of validation requests

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.

Figure 8 shows the throughput of the Leader and FSM Managers - the number of validate requests handled per second and the scalability of the approach.

Average Event Time Performance Measurement Measuring the average event time performance we considered a standard system containing a Leader, three Backups, m FSM Managers and 52 Components per FSM Manager (50 temperature sensors, 1 smoke sensor and 1 sprinkler). 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 9). This would place increased load on the Leader.

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.

Fault Tolerance While previous performance tests had considered GOANNA in simple initialisation and computation phases, failures were introduced to mea-

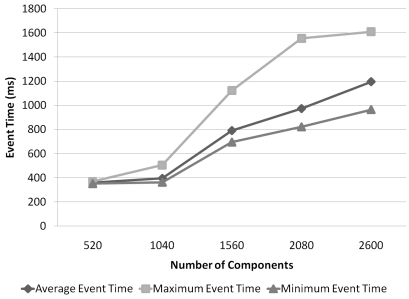

Fig. 9. Base Average Event Time Performance

Fig. 10. Kill Performance

sure the capacity for GOANNA to deal with the addition (booting) and removal (killing) of FSM Managers and Components at runtime.

Three Backups including a single leader 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.

- **Kill.** In each experiment a set of 50 FSM Managers were seen to operate normally. We introduced failures by killing the FSM Manager process on individual hosts. Given the kill scenario, once a FSM Manager was removed from the GOANNA system it was *not* restarted. Reduced FSM Manager load on the Leader was seen to improve the performance of GOANNA, with reduced AET after a FSM Manager had been killed. Figures 10 and 12 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 10 illustrates the Maximum, Minimum and Average Event Time occurring where FSM Managers are removed from the GOANNA system. The Leader was seen to handle the removal and time-out of FSM Manager messages gracefully.
- **Kill-Reboot.** GOANNA was seen to perform consistently and normally where FSM Managers were first killed and then rebooted. As rebooting restarted, FSM Manager processes were seen to not adversely affect the AET of the overall system. AET had limited fluctuation between 1143 ms and 1271 ms for increasing FSM Managers. Figure 11 shows the Maximum, Minimum and Average Event Time occurring where FSM Managers are killed and then rebooted.

Comparison. GOANNA is seen to continue to execute normally where FSM Managers were removed from the system (Figure 12). A Leader's 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. In the instance where FSM Managers were rebooted the system performance was seen to behave as if no failure had occurred in the system - results mimicked a faultless system.

Fig. 11. Kill-Reboot Performance

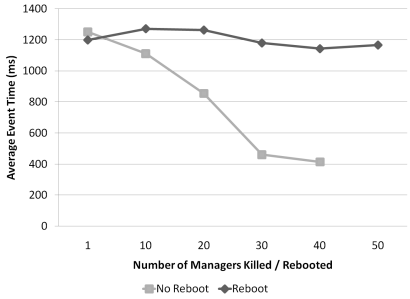

Fig. 12. Comparison of Performance: Kill and Kill-Reboot

4.1 Summary

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. Performance degradation for systems exceeding 2600 components produced a shallow derivative in terms of AET performance loss (Figure 9). Thus, 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 sensor deployment.

5 Related work

Various techniques have been developed in order to generate a distributed implementation from a logically centralised specification. In [14] the authors use an aspect-oriented approach in order to automatically generate the global behaviour. 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 also be used in property verification. In [15] 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 [16] 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 [17] the authors present a workflow engine that simulates distributed execution by migrating the workflow instance (specification plus run-time data) between execution nodes. In [18] 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 fault and supports the introduction of new component instances at runtime.

References

1. Guerraoui, R., Rodrigues, L.: Reliable Distributed Programming. Springer (2006)
2. Oppenheimer, P.: Top-down network design. Cisco system inc. (2004)
3. Schroeder, B.A.: On-line monitoring: A tutorial. IEEE Computer **28** (1995) 72–78
4. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: PODC '07. (2007) 398–407
5. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: OSDI. (2006)
6. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: ASE. (2002) 3–12
7. Penna, G.D., Magazzeni, D., Intrigila, B., Melatti, I., Tronci, E.: Automatic generation of optimal controllers through model checking techniques. In: ICINCO-ICSO. (2006) 26–33
8. Mostarda, L., Marinovic, S., Dulay, N.: Distributed orchestration of pervasive services. In: IEEE AINA. (2010)
9. Pediaditakmois, D., Mostarda, L., Dong, C., Dulay, N.: Policies for Self Tuning Home Networks. In: IEEE POLICY 09. (2009)
10. Lamport, L.: Paxos made simple, fast, and byzantine. In: OPODIS. (2002) 7–9
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21** (1978) 558–565
12. Ben-Ari, M.: Principles of Concurrent and Distributed Programming (2nd Edition). Addison-Wesley (2006)
13. Mostarda, L., Ball, R., Dulay, N.: Distributed fault tolerant controllers. Technical report, Depart. of Computing Imperial College London (2010)
14. Cao, F., Bryant, B.R., Burt, C.C., Raje, R.R., Olson, A.M., Auguston, M.: A component assembly approach based on aspect-oriented generative domain modeling. Electr. Notes Theor. Comput. Sci. **114** (2005) 119–136
15. Chen, F., Rosu, G.: Towards monitoring-oriented programming: A paradigm combining specification and implementation. Electr. Notes Theor. Comput. Sci. **89** (2003)
16. Inverardi, P., Mostarda, L., Tivoli, M., Autili, M.: Synthesis of correct and distributed adaptors for component-based systems: an automatic approach. In: ASE. (2005) 405–409
17. Montagut, F., Molva, R.: Enabling pervasive execution of workflows. 2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing (2005)
18. Sen, R., Roman, G.C., Gill, C.: CiAN: A Workflow Engine for MANETs. Coordination Models and Languages (2008) 280–295