2010 24th IEEE International Conference on Advanced Information Networking and Applications

Distributed Orchestration of Pervasive Services

Leonardo Mostarda, Srdjan Marinovic, Naranker Dulay
Department of Computing
Imperial College London
Email: {lmostard, srdjan, nd}@imperial.ac.uk

***Abstract*—Pervasive systems are increasingly being designed using a service-oriented approach where services are distributed across wireless devices of varying capabilities. Service orchestration is a simple and popular method to co-ordinate web-based services but introduces a single point of failure and lacks the flexibility to cope with the greater variability of pervasive environments. Choreography in contrast advocates explicitly modelling systems as interacting peers that conform to rules of interaction. Choreography offers greater reliability and flexibility but leads to systems that are much harder to validate. In this paper we describe a novel intermediate approach, where given a logically centralised service orchestration, we automatically generate a distributed implementation that correctly enforces the orchestration behaviour. Our system handles all the synchronisation and consensus issues and ensures correctness. The system also incorporates a number of abstractions for grouping pervasive peers and coordinating pervasive peer-to-peer interactions.**

***Keywords*-Pervasive systems, distributed systems, choreography, workflows.**

I. Introduction

Coordinating services for pervasive environments presents new software engineering challenges where traditional service-oriented approaches [1] need to be augmented or replaced by *proactive* and *autonomous* services that co-operate to achieve to overall goals. Sense and react systems [2], unmanned autonomous vehicles undertaking a rescue mission [3] are two examples of systems where services are distributed across wireless devices of varying capabilities operating in difficult environments but whose behaviour must be correctly coordinated in order to achieve the overall goals of the system.

Workflows are a well-established paradigm for organising business processes in enterprises. They have also been proposed for pervasive environments [4], [5], [6], [7], [8] to build applications that discover and orchestrate pervasive services.

Although orchestration offers a relatively easy way to build a service-based application, its overheads and centralised execution are not well-suited to pervasive environments, where there is a need to distribute the service orchestration behaviour to cope with failures and handle varying numbers of services and actors. Some of these concerns can be addressed by choreography-based workflows, which advocate that programmers explicitly model distributed systems as interacting peers that conform to rules of interaction. Choreography potentially offers greater scalability and reliability, but choreography-based applications [9] are harder to validate and develop as they require programmers to implement application-dependent and often subtle synchronisation and consensus protocols [10], [11], [12].

In this paper we describe a novel intermediate approach that automatically generates a *distributed*, *scalable* and *fault tolerant* choreographed implementation from a logically centralised orchestration specification. We have implemented this approach for a small workflow language based on BPEL [13] called CHOREO and have tailored it for some of the requirements of pervasive services. Workflows in CHOREO are translated into partitioned finite state machines and executed locally at service endpoints. The CHOREO runtime system ensures correct synchronisation and consensus. In order to validate the approach we present a service-based fire alarm workflow system.

Section 2 of the paper presents an overview of how the proposed approach for workflow specification and execution, is employed in a fire alarm case study. Section 3 details the CHOREO workflow language and Section 4 discusses how exception handling is specified. Section 5 discusses the translation of a CHOREO specification into a finite state machine. Section 6 describes finite state machines' decentralised execution and section 7 summarises the performance of decentralised execution. Finally, section 8 compares the presented work with existing related works in this area while section 9 provides a conclusion and outlines future work.

II. Approach overview

In this section we describe a small service-based fire alarm application and how a choreographed implementation is automatically generated from the specification of a centralised orchestration in CHOREO.

Our system model assumes that applications are built using components that provide and require services. Workflows are used to define the correct orchestration of component interactions and can proactively invoke component services as well as intercept service interactions among components. Service interception can be used to check that the components are interacting correctly, or to adapt the behaviour and structure of the system to changing context.

1550-445X/10 $26.00 © 2010 IEEE
DOI 10.1109/AINA.2010.100

```
workflow fireAlarm(set tempSet:Temperature, set smokeSet:
    Smoke, set sprinklerSet:Sprinkler)
  while (true)
    flow   -- do activities in parallel
      invoke tempSet.getTemp() => temp
      invoke smokeSet.getLevel() => smoke
    wait notification(temp)
      if temp > 50
        wait notification(smoke)
          if smoke > 30
            invoke sprinklerSet.waterOn() => oneway
            pick   -- wait for first event
              wait call(sprinklerSet, *, start)
              wait timeout(1000)
                invoke sprinklerSet.alarm() => oneway
          else
            invoke sprinklerSet.waterOff() => oneway
      else
        invoke sprinklerSet.waterOff() => oneway
```

Figure 1. Fire alarm workflow in CHOREO

In our fire alarm workflow (See Figure 1) we have three components. The `Temperature` and `Smoke` components provide the services `getTemp()` and `getLevel()` to obtain temperature and smoke readings, respectively. The actuator component `Sprinkler` provides the services `waterOn()` and `waterOff()` to switch the sprinkler on or off. There is also the `Driver` component that is bound to Sprinkler components that provides the `start()` and `stop()` services to open and close the water flow. The basic requirement is that the sprinkler must provide water only when both the temperature and smoke readings exceed a threshold.

The workflow is defined over sets of components. A set groups component instances of the same type and provides an abstraction to separate the workflow specification from the knowledge of how many components are available at any given moment in the environment. For example, in the fire alarm system we want to obtain smoke and sensor readings from available sensors without knowing their number in advance.

`Invoke` activities implement asynchronous service calls. For instance `invoke tempSet.getTemp() => temp` is used to invoke the service `getTemp()` on one component instance belonging to `tempSet`. `temp` is a notification object where replies to the invocation will be returned. The keyword `all` can be used to broadcast the invocation to all component instances in the set.

`Wait` activities are used to wait for notifications of replies and also to intercept service invocations. For instance, `wait notification(temp)` is used to wait for the result of the aforementioned `invoke` while `wait call(sprinklerSet, *, start)` can be used to wait for a sprinkler to call the `start` service on any component (denoted with *). We emphasise that `invoke` is a best effort activity that does not guarantee the service call is delivered. The wait activity can be used to catch any failures that arise from the invocation.

System configuration. CHOREO system configurations

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

  workflow wf:fireAlarm(t, sm, sp);
```

Figure 2. Fire alarm configuration

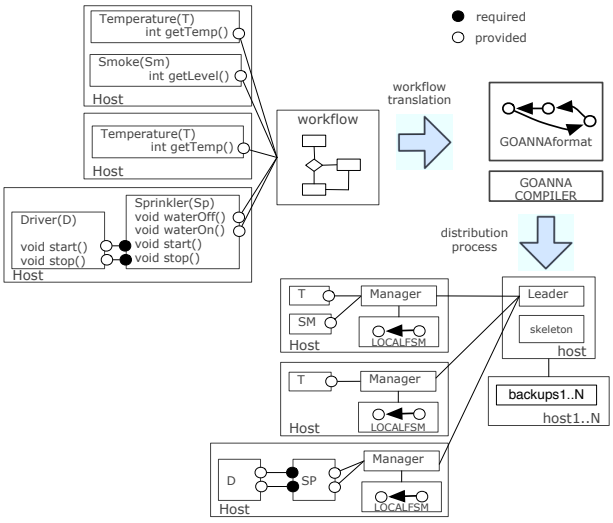

Figure 3. Compilation and distribution

specify both workflows and component sets. 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 2, the three sets: `t`, `sm` and `sp` will group all components of types `Temperature`, `Smoke` and `Sprinker` respectively running on floor `floor` of the building. CHOREO workflows and configurations can be instantiated multiple times and discovered components can be members of more than one set. For example, the fire alarm configuration can be instantiated for each floor.

Compilation and Distribution. In Figure 3 we show the stages for workflow compilation and distribution. In the former, each workflow is translated into a finite state machine form. This is passed to the GOANNA finite state machine compiler [14] which verifies several properties and generates a distribution implementation. Properties checked include correct ordering between an invoke and its reply, progress and safety (e.g., a waterOn call always follows a high temperature and smoke detection). The distribution stage decomposes each finite state machine (workflow) into a collection of local ones and a skeleton. Local state machines are passed to the finite state machine (FSM) manager at each host while the skeletons are passed to a leader process running on some host and its backups running on other

hosts. Managers and the leader implement a version of the Paxos with Steady State consensus protocol to ensure the correctness of the distributed workflow implementation.

III. CHOREO LANGUAGE

CHOREO is based on BPEL [13] but is designed to be used for the orchestration of pervasive services. The syntax for CHOREO workflows is given in Figure 4. Note: Blocks are grouped lexically using indentation instead of syntactically (like in Python). The language supports the following workflow activities:

- **invoke** - this activity calls a service on a specified set. By default, invoke calls one randomly selected service from the set. The random selection is done using a fair coin toss and is implemented by the GOANNA execution platform. Replies to this invocation will be made available to the workflow through a *notification* object. By specifying the keyword **all**, the invocation will be broadcast to all the services in the set and the notification object will receive all replies. The keyword **oneway** can be used if the service does not return a reply or if the workflow is not interested in the reply.
- **wait** - this activity is used to block the execution of a workflow for the following cases: waiting for a reply to an invocation (*wait notification*), waiting for a service to make a call to another service (*wait call*), waiting for a service to receive a request from another service (*wait request*), and waiting for a specific time duration (*wait timeout*).
- **pick** selects the first wait activity that completes and executes its associated activity sequence.
- **seq** executes a list of activities in sequence, while **flow** executes a list of activities in parallel. **flow** assumes that the activities are mutually-independent, and that there is no need for synchronisation between them.
- **if**, **while** and **foreach** implement conditional choice and iteration.

It is worth noting, that we have departed from traditional workflow concepts where invoke is performed on a particular service and the result notification is strictly linked to the service. We believe that set-based invocations are more appropriate for pervasive environments where there can be a large number of services offering the same operations and where all or some results are needed. For example, this approach is often used to orchestrate sense-and-react applications.

Monitoring of service interactions also plays an important part in adapting an application's behaviour. To this end, CHOREO extends the semantics of a typical *wait* activity to observe interactions between different services, using *wait call* and *wait request*. *wait call* allows the workflow to monitor the invocation of a service call at a component, whereas *wait request* allows the workflow to monitor the arrival of a service request at a component.

```
workflow =  workflow workflowname "(" formalparams ")"
     sequence

sequence = [seq] (flow | pick | wait | invoke | if | while
      | foreach | skip)+

flow = flow sequence+

pick = pick wait+

wait = wait event sequence [exception sequence]

event = notification "(" notificationname ")"
  | call "(" fromsetname, tosetname, servicename ")"
  | request "(" fromsetname, tosetname, servicename ")"
  | timeout "(" timeexpression ")"

invoke = invoke [all] setname "." servicename "(" params
     ")" "=>" (notificationname | oneway)

if = if booleanexpression sequence [else sequence]

while =  while whileexpression sequence

foreach = foreach foreachexpression sequence

skip = skip

params, names, expressions omitted
```

Figure 4. CHOREO syntax

IV. HANDLING FAILURES IN A WORKFLOW

During the execution of a workflow there is always a possibility of failures. CHOREO workflows can handle invocation failures and *wait* timeouts.

The invoke activity represents an asynchronous invocation (as opposed to BPEL's invoke activity which represents a synchronous invocation) and hence it does not raise an exception if the services in the invocation set are unreachable or if the set is empty. The underlying GOANNA layer will attempt to discover services and deliver the service call to them. The corresponding *wait notification* will block for a specified *timeout* specified per set, in a configuration file. If GOANNA fails to discover or reach any services for an invoked set, the corresponding *wait notification* activity will receive an exception. Furthermore, if the notification is not received within this *timeout* period and exception will also be raised.

```
invoke set1.service1() => n1
wait notification(n1)
  invoke set2.service2() => oneway
exception
  invoke set2.service3() => oneway
```

The *pick* activity can be used to specify the *timeout* period for more than one wait as in the following example:

```
invoke set1.service1() => n1
invoke set2.service2() => n2
pick
  wait notification(n1)
    skip
  wait notification(n2)
    skip
  wait timeout(1000)
    invoke set3.service3() => oneway
```

V. Generating a State Machine Representation of a Workflow

CHOREO workflows are compiled into a GOANNA finite state machine by the CHOREO compiler. The GOANNA finite state machine is then decomposed into a collection of local state machines that can be distributed and executed in a completely distributed way. Each GOANNA finite state machine consists of: (1) **Set of states** S, where every state is a member of $\mathbb{N}_0$. (2) **Set of transitions** T, where a transition is a tuple defined as: $[S_{start}, S_{end}, token]$.

A *token* is an element of the following set: $\{invoke, event, exception, cond, !cond\}$. The *invoke* token corresponds to executing an *invoke* activity. The *event* and the *exception* tokens are used to represent the *wait* activity, as shown in the Figure 6. The *event* token represents the start of the sequence associated with a *notification*, *call*, *request* or *timeout* part of the *wait*'s specification. Whereas, the *exception* token represents the start of the *exception* sequence. The tokens *cond* and *!cond* are used to express transitions based on whether an *if*'s and *while*'s condition has been evaluated as true or false.

The algorithm for translating a CHOREO workflow into a finite state machine is:

```
1  generate(Sequence s) return fsm
2    Set states = {0};
3    Set transitions = {};
4
5    forall activity in s
6      last = states.last; -- state with the greatest n
7      case activity is
8        invoke : s1 = states.addNew;
9          transitions.add([last, s1, activity]);
10
11       wait : s1 = states.addNew;
12         s2 = states.addNew;
13         transitions.add([last, s1, wait.event]);
14         transitions.add([last, s2, wait.exception]);
15         connect(s1, generate(wait.eventSeq));
16         connect(s2, generate(wait.exceptionSeq));
17
18       if : s1 = states.addNew;
19         s2 = states.addNew;
20         transitions.add([last, s1, if.cond]);
21         transitions.add([last, s2, if.not_cond]);
22         connect(s1, generate(if.condSequence));
23         connect(s2, generate(if.not_condSequence));
24
25       foreach :
26         for 1 to foreach.N
27           connect(last, generate(foreach.sequence));
28           last = states.last;
29         end
30
31       while.withPredicate :
32         s1 = states.addNew;
33         seq = generate(while.sequence);
34         seq.lastState(last);
35
36         transitions.add([last, s1, while.cond]);
37         connect(s1, seq);
38         s2 = state.addNew
39         transactions.add([last, s2, while.not_cond]);
40
41       while.isTrue :
42         seq = generate(while.sequence);
43         seq.endState(last);
44         connect(last, template);
45
46       flow :
47         connect(last, interleave(flow.sequences))
48     end
49   end
50   return [states, transitions];
51 end
```

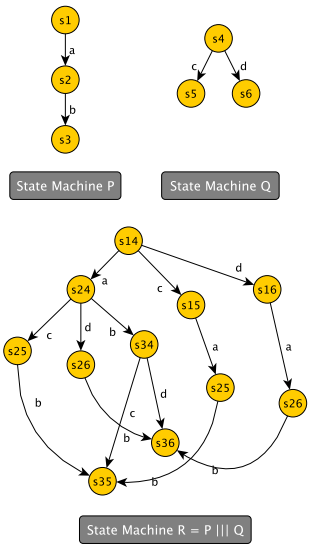

Figure 5. Flow interleaving example

The workflow is represented as a sequence activity which is passed as the parameter for the initial invocation of the *generate* function. The state machine is constructed with an initial state *0*. In case the next activity in the sequence is an *invoke*, a new state is added to *states* and the *invoke* activity is used as token to create a transition between the last state and the newly created one.

For other activities, the algorithm simply creates the structures corresponding to their respective templates which are shown in Figure 6. The *connect* function replaces the initial state (*start* in templates) of the state machine, given as the second parameter, with the state, given as the first parameter. For *while* templates it is not enough to simply connect the *last* state of the currently generated state machine with the while's sequence, one must also *loop back* to the last state of the currently generated state machine. In order to do this, before calling the *connect* function, the algorithm generates the while's sequence and puts the current last state as the last state of the generated sequence (effectively looping back).

However, there is no template for the *flow* activity since this structure interleaves transitions from all the sequences that it contains. It is assumed that there is no synchronisation between the finite state machines and thus all possible traces are produced (Figure 5). The interleaving algorithm is based on the *interleaving* operator ($|||$) from the CSP language [15].

Figure 7 shows the state machine obtained by translating

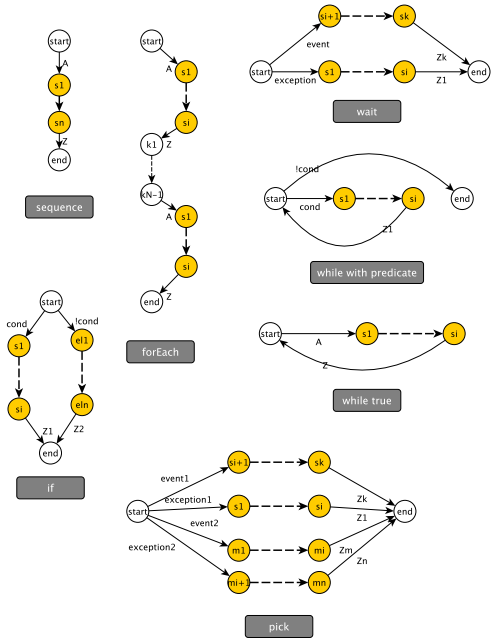

Figure 6. State Machine-based Templates

the case study presented in Figure 1. The numbers between the "[" and "]" brackets represent the line number that a particular token represents. Figure 8 shows the GOANNA version of the state machine. This is defined by a list of epsilon moves and events each followed by its transition rules. A transition rule has the form `s1 - s2: condition -> action`[1]. This is applied when its event is observed, the current state of the global state machine is `s1` and the `condition` is true. The rule's `action` is performed and the state updated to `s2`. An action that contains a `signal to c in setName serviceName` primitive defines a call to the service `serviceName` on a component belonging to the set `setName` and `c` is the name assigned to the component chosen for the call.

Wait events are expressed using the syntax `serviceName on set1 to set2` which denotes a serviceName response sent by a component belonging to the set `set1` to a component in `set2`. For instance `getTemp on tempSet to` $\star$ denotes a getTemp response event sent by a component belonging to tempSet to a component in * (i.e., an unknown set).

A. State machine decomposition

The GOANNA compiler decomposes a workflow's finite state machine into a set of local ones (one for each set), plus a skeleton. A local state machine contains all events local to its set plus some epsilon moves. The epsilon moves

[1]The initial state is the first listed

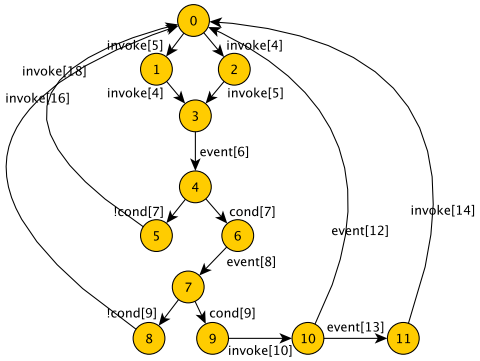

Figure 7. Finite state machine for Fire Alarm Workflow

```
global fsm fireAlarm(set Temperature tempSet,set Smoke smokeSet,
                     set Sprinkler sprinklerSet){
  on epsilon
     0-1:   -> signal to temp in tempSet getTemp();
     0-2:   -> signal to smoke in smokeSet getLevel();
     1-3:   -> signal to smoke in smokeSet getLevel();
     2-3:   -> signal to temp in tempSet getTemp();
     5-0:   -> signal to a1 in sprinklerSet waterOff();
     8-0:   -> signal to a2 in sprinklerSet waterOff();
     9-10:  -> signal to a3 in sprinklerSet waterOn();
     11-0:  -> signal to a4 in sprinklerSet waterOn();
     4-6:  (component.name=="temp")&&(event.result>50) -> {}
     4-5:  (component.name=="temp") -> {}
     7-9:  (component.name=="smoke")&&(event.result>50) -> {}
     7-8:  (component.name=="smoke") -> {}
  getTemp  on  tempSet to *
     3-4:  (component.name=="temp") -> {}
  getLevel  on  smokeSet to *
     6-7:  (component.name=="smoke") -> {}
  start  on  sprinklerSet to *
     10-0: -> {}
  on  timeout(1000)
     10-11: -> {}
}
```

Figure 8. GOANNA version of FSM

can be either those where no condition is defined (i.e., any component can perform them) or the ones that the set relates to. The skeleton contains all timeout rules. In Figure 9 we show the local state machine related to the set `tempSet`. It contains the rules related to the service $getTemp()$ plus the related epsilon moves.

```
global fsm fireAlarm(set Temperature tempSet,set Smoke smokeSet,
                     set Sprinkler sprinklerSet){
  on epsilon
     0-1:   -> signal to temp in tempSet getTemp();
     0-2:   -> signal to smoke in smokeSet getLevel();
     1-3:   -> signal to smoke in smokeSet getLevel();
     2-3:   -> signal to temp in tempSet getTemp();
     5-0:   -> signal to a1 in sprinklerSet waterOff();
     8-0:   -> signal to a2 in sprinklerSet waterOff();
     9-10:  -> signal to a3 in sprinklerSet waterOn();
     11-0:  -> signal to a4 in sprinklerSet waterOn();
     4-6:  (component.name=="temp")&&(event.result>50) -> {}
     4-5:  (component.name=="temp") -> {}
  getTemp  on  tempSet to *
     3-4:  (component.name=="temp") -> {}
}
```

Figure 9. The local state machine for tempSet

VI. Runtime Implementation

As we have seen previously, our architecture (see Figure 3) is composed of an *FSM manager* local to each host, a *leader* and set of *backups*. The *leader* (see Figure 10) holds the workflow skeleton and the current global state of each workflow instance. A local timer is used to notify time out events. These events can change a state when a timeout transition rule is applied. A structure is used to maintain for each set all hosts containing at least one component belonging to the set. This is used to implement the `invoke` when a broadcast is required. A lock is used to avoid race conditions during state updates, while the key denotes the protocol instance. Each *FSM manager* has a loader that searches a local directory for new components. When a new component is found, the loader loads its code and evaluates all sets the components belongs to. This is performed by matching the component type with each set definition and verifying all constraints defined by the `where` clause, e.g., current location and host name. When a new set is populated the related local state machine is loaded and the leader updated. The *backups* are managers that hold, for each state machine instance, a copy of the global state. This allows recovery in case of leader failures.

FSM managers and the leader implement the Multi-Paxos with Steady State protocol [12]. The basic idea is that the leader and a majority of backups hold the last updated states for each state machine instance. Managers have their local states but these can be out of date. Managers propose a new global state based on its local ones. If these are out-of-date, the manager will get synchronised by the leader and it can retry with the updated states. When managers want to apply transition rules, at the same time, the leader will randomly pick one of them up. This validates the constraints imposed by each global state machine, that is when a state is exited by two transitions only one of them can be applied at the same time. The chosen manger applies the rules (i.e., execute the related actions) and updates the new global states on the majority of backups. We use the Multi-Paxos with Steady State protocol in order to propagate each new state through the backups. In particular, this is enriched with timeouts and additional messages in order to synchronise the managers and to correctly execute actions of the state machines. Paxos protocols are normally described using client, acceptor, learner, and leader[2] roles. In our implementation the client, acceptor and learner roles are included in the FSM manager.

In Figure 10 we show the message flow for a successful protocol execution of the fire alarm system. Suppose that a FSM manager observes an event `start` yellow from a sprinkler component and its FSM instance is in state 10. Suppose further that the leader also has the state of the FSM instance equal to 10 and that the instance has not been

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

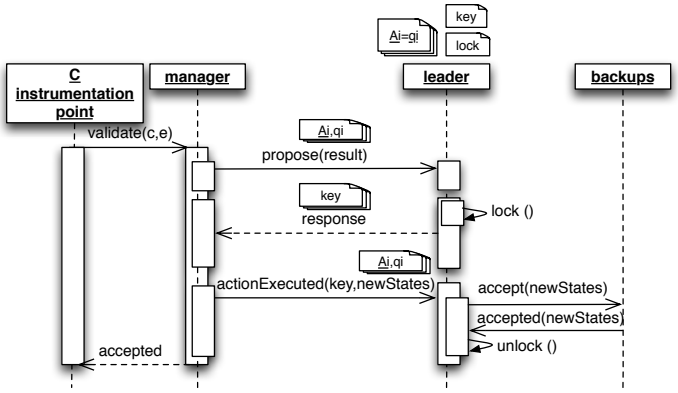

Figure 10. Run time execution

locked. Since the FSM manager can match `start` using the rule `10-0: {} -> {}` (this has been projected to the local sprinklerSet FSM from the global FSM of Figure 8) it starts a protocol instance sending a `propose(result)` message to the leader. This contains the state machine instance and the related state of acceptance (i.e., $s = 10$). The leader compares the received state $s = 10$ with the current state $sk = 10$. Since they are the same and its FSM has not been locked by another manager the leader generates a `response` message which contains a new key which promises to the manager the lock on its state machine instance. The manager receives the request, performs the local action (none in this case) and performs all epsilon rules following the new state 0 which is either the chain 0-1 and 1-3 or 0-2 and 2-4. These cause the signal on temp and smoke component instances. After all epsilon moves have been performed the manager sends back to the leader an $actionExecuted(key, newStates)$ reply where key is the protocol instance key and newStates is either 3 or 4. The leader receives the reply and checks the existence of the key. In the case that the key already exists it deletes the key, updates the skeleton state, updates all backup managers and unlocks the skeleton. The update of the backups require that the majority of them correctly update the new state.

Different exceptions can be raised during protocol execution. A *manager out of sync* exception is received by a FSM manager if it proposes states, which are out-of-date with respect to the leader. In this case the FSM manager is updated. A *locked* exception is received by a FSM manager when the skeleton has been already locked. A *manager timeout* exception is raised on the leader side when a manager receives the permission to apply a rule but does not respond with an $actionExecuted$ message. This can be a consequence of a manager fault or an action that is slow to execute. In this case the leader deletes the key and unlocks the skeleton. Managers always retry after an exception until

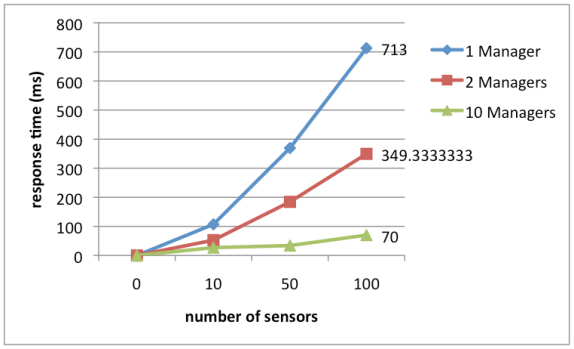

Figure 11. Response Time (ms)

this is propagated at the workflow level. In our protocol, special managers are entrusted to detect a leader failure. More specifically when the leader is no longer available managers detect it, a new leader is elected and all correct global states recovered from the backup managers.

VII. Performance

A distributed implementation of a logically centralised state machine specification brings overheads in terms of network traffic and synchronisation time. In this section we examine the time and memory overheads of our approach.

In order to study the overall performance of our distributed implementation, we looked at the time it takes to perform a state machine transition and also the 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.

There are three versions of our underlying GOANNA platform, one for Java 1.5, one for C, and one for TinyOS/NesC. For this paper we report on using the GOANNA for Java 1.5 version running on a 100 Mbit network of ten Pentium IV 3.2GHz machines each with 2GB of RAM running the Linux operating system. This setup has a fast CPU and network but high overheads in terms of memory consumption and RMI of Java 1.5.

We performed the evaluation on three different configurations to show the effect of increasing distribution: (A) one manager and one leader; (B) three managers and a leader; (C) ten managers and a leader. For each configuration we ran the same three experiments. We ran systems with 10, 50 and 100 smoke and temperature sensor components. More specifically 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, a third of the sensors were run on each host. For configuration C, a tenth of the sensors were run on each host.

Figure 11 shows a manager's response time for all three configurations. Each value was obtained by running the experiments for 10 minutes and calculating the average of all

Component type	Code (bytes)	Data (bytes)
Manager	1120	58
Leader	890	34
Temperature Component	4530	40
Temperature local state machine	870	12

Figure 12. TinyOS 2 Code and data sizes

transition execution times. For example, in the case where 100 sensors were running at the same time, the results were as follows: 713 ms for configuration A ; 349 ms for configuration B and 74 ms configuration C. This shows that the protocol scales linearly when components are distributed across different hosts. The critical bottleneck is the manager and not the leader. The 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. We also evaluated the throughput for the case of a single manager and a leader. In this case, the leader received an average of 86 requests per second; each request is 2 bytes.

For Java 1.4 the heap and data memory consumption is 900 KB for a manager, and 489 KB for the leader (in the worst case). The local state machine descriptions averaged about 1.5 KB in size. Most of the overheads of this are due to Java. In contrast, our implementation for TinyOS2 running on TmoteSky sensors is much more lightweight (see Figure 12).

VIII. Related work

Service orchestration in pervasive environments has been explored by Berger et. al [5] and Ranganathan [4]. Their work looked at how pervasive web services can be orchestrated from a workflow engine running on a user's mobile device. Similar approaches include Pajunen et. al [7] and Hackmann et al. [8]. In our work, we address a wider challenge, were orchestration behaviour is distributed among the pervasive services and infrastructure. An earlier example of this, is ORBWork [16], which used CORBA components to implement services and ORBWork nodes to invoke these services. The order of invocations was statically distributed to all nodes. A similar approach was more recently implemented in SwinDew [17] and CiAN [18]. None of these systems handles dynamic systems or failures nor do they automatically decompose orchestrations into choreographed execution. A very different approach to distributed execution is presented in Montagut et al. [6] where workflow instances migrate from one workflow node to another. Similarly in [19] Manolescu advocates the use of continuations to effectively freeze a workflow instance and restart it in a different execution environment. However, even though the migrating workflow operates over multiple execution nodes over a period of time, it does not handle the failures of the nodes that are currently executing the workflow. Furthermore, the migration, by itself, does not

solve the synchronisation issues when the workflow is being executed by multiple nodes at the same time. As the previous sections have indicated, these issues represent the core goals of our distribution model.

IX. Conclusions and future work

In this paper we have presented a novel approach for the distributed orchestration of pervasive services where logically centralised workflows are automatically compiled into choreographed implementations using synchronised finite state machines and an underlying consensus protocol. Our workflow language is based BPEL but extended to support communication primitives more suited to pervasive environments. For our future work, we plan to support the concept of a human-centric task as part of the workflow and to extend the language to support goal-driven workflows with more flexible ordering of activities. In addition to these proposed extensions, we are currently investigating suitable transaction and rollback extensions to our finite state machine execution model.

Our future work effort regarding the implementation and its testing will be to assess the energy-consumption of the computations needed to run a CHOREO workflow in wireless sensor networks.

X. Acknowledgements

This research was supported by UK EPSRC research grant EP/D076633/1 (UBIVAL) and EU FP7 research grant 213339 (ALLOW).

References

[1] B. Benatallah and H. R. M. Nezhad, "Service oriented architecture: Overview and directions," in *Lipari Summer School*, 2007, pp. 116–130.

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

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

[4] A. Ranganathan and S. McFaddin, "Using workflows to coordinate web services in pervasive computing environments," *Web Services, IEEE International Conference on*, 2004.

[5] S. Berger, S. McFaddin, C. Narayanaswami, and M. Raghunath, "Web services on mobile devices-implementation and experience," *Mobile Computing Systems and Applications, 2003. Proceedings. Fifth IEEE Workshop on*, pp. 100–109, Oct. 2003.

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

[7] L. Pajunen and S. Chande, "Developing workflow engine for mobile devices," *11th IEEE International Enterprise Distributed Object Computing Conference*, 2007.

[8] G. Hackmann, M. Haitjema, C. Gill, and G.-C. Roman, "Sliver: A bpel workflow process execution engine for mobile devices," *Service-Oriented Computing –ICSOC 2006*, pp. 503–508, 2006.

[9] M. Younas, I. Awan, R. Holton, and D. A., "Duce: A p2p network protocol for efficient choreography of web services." in *AINA*, 2006, pp. 839–846.

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

[11] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC '07*, 2007, pp. 398–407.

[12] L. Lamport, "Paxos made simple, fast, and byzantine," in *OPODIS*, 2002, pp. 7–9.

[13] K.-M. Chao, M. Younas, and N. Griffiths, "Bpel4ws-based coordination of grid services in design." in *Computers in Industry*, 2006, pp. 778–786.

[14] L. Mostarda, D. Sykes, and N. Dulay, "A State Machine-Based Approach For Reliable Adaptive Distributed Systems," in *7th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, March 2010.

[15] C. Hoare, *Communicating Sequential Processes*, ser. Prentice Hall International Series in Computing Science. Prentice Hall, 1985.

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

[17] J. Yan, Y. Yang, and G. K. Raikundalia, "Swindew-a p2p-based decentralized workflow management system," *IEEE Transactions on Systems, Man, and Cybernetics Part A - Systems and Humans*, 2005.

[18] R. Sen, G.-C. Roman, and C. Gill, "Cian: A workflow engine for manets," *Coordination Models and Languages*, pp. 280–295, 2008.

[19] D. A. Manolescu, "Workflow enactment with continuation and future objects," *SIGPLAN Not.*, vol. 37, no. 11, pp. 40–51, 2002.