Pervasive Healthcare using Self-Healing Agent Environments

Stefano Bromuri, Michael Ignaz Schumacher, Kostas Stathis

Abstract Pervasive healthcare systems (PHSs) are required to be constantly available to the patients accessing them. To address this issue, in this paper we present an agent-based PHS that self-heals one or more of its parts when a service disruption happens. We propose a multi-agent system (MAS) approach that utilises coordination, planning and the notion of agent environment to create a distributed system capable to heal itself even if 50% of the system is not functioning due to external causes.

Key words: Multi-Agent Systems, Self-Healing, Planning, Pervasive Healthcare.

1 Introduction

Pervasive Healthcare [12] focuses on bringing healthcare everywhere, breaking the boundaries of hospital healthcare. In [12] Varshney defines Pervasive Healthcare Systems (PHSs) as complex systems where multiple components interact to allow large scale monitoring of physiological data of heterogeneous patients. Two main limitations affect existing PHSs: (a) failures of the distributed system are never taken into consideration and (b) the system topology is statically defined. Consequently, PHSs need fault tolerance mechanisms as system downtimes may be dangerous for patients relying on them. Space-based redundancy [5], a practice that improves the resilience of an infrastructure by replicating components, should be avoided as it is an expensive practice for distributed systems like PHSs. In this paper we address

Stefano Bromuri, Michael Ignaz Schumacher
Business Information Systems, University of Applied Sciences Western Switzerland, TechnoArk 3, CH-3950, Sierre, Switzerland e-mail: {stefano.bromuri,michael.schumacher}@hevs.ch

Kostas Stathis
Department of Computer Science, Royal Holloway University of London, Egham Hill, EGHAM, TW20 0EX, e-mail: kostas.stathis@rhul.ac.uk

PHSs fault tolerance via the self-healing paradigm [7] that focuses on time-based redundancy [5], which, instead of replicating components, replicates components behaviours to ensure that if a component fails an existing component can substitute it. Mikic-Rakic et al. in [9] identified the following properties as necessary for self-healing systems: adaptability, dynamicity, awareness, observability, autonomy, robustness, distributability, mobility and traceability.

Multi agent systems (MASs) [15] represent a valid abstraction to model such systems and fulfil the self-healing paradigm requirements. In particular, the adoption of MASs facilitates the transition from a centralised computing model to a decentralised one where thousands of autonomous agents interact to achieve a common goal. Moreover, the concept of agent environment [14] has been accepted as a useful abstraction to mediate the interaction between agents and to model how the agents perceive resources and interfaces that they utilise in their interaction.

In [1] we proposed a PHS based on a distributed agent environment built on the GOLEM[1] platform [2], to support intelligent agents monitoring patients affected by diabetes. In this system we mapped the agent environment represented as a distributed rectangular grid to a real environment representing a city. In this paper we extend such a PHS by defining a novel coordination and planning algorithm that agents use to detect faults and recover the system functionalities. The contributions of this paper are: a) we introduce a practical approach based on agents to handle fault tolerance in PHSs; b) we split responsibilities between the agents and the agent environment thus simplifying the behaviour of the agents to fulfil the requirements of self-healing systems; c) we illustrate how a compact declarative specification for the agents behaviour can deal with multiple failures of the agent environment in parallel.

The rest of this paper is structured as follows: Section 2 describes the self-healing system we developed in terms of its main components; Section 3 evaluates our approach and discusses relevant related work; finally Section 4 concludes this paper and presents future directions.

2 Extending GOLEM with Self-Healing Procedures

In this paper we extend the PHS presented in [1] with self-healing procedures. The system presented in [1] is based on the GOLEM agent platform [2] whose main abstractions are agents, cognitive entities, objects, reactive entities available to the agents as resources, and containers, declaratively programmed distributed spaces where agents and objects are situated, defining a distributed agent environment. Fig. 1 shows the conceptual architecture of our extended system in relationship to its distributed topology.
We introduced an infrastructure agent in the containers of the distributed agent environment and mediation rules to mediate the messages exchanged by the agents, to

[1] GOLEM stands for Generalised Onto-Logical Environments for Multi-Agent Systems.

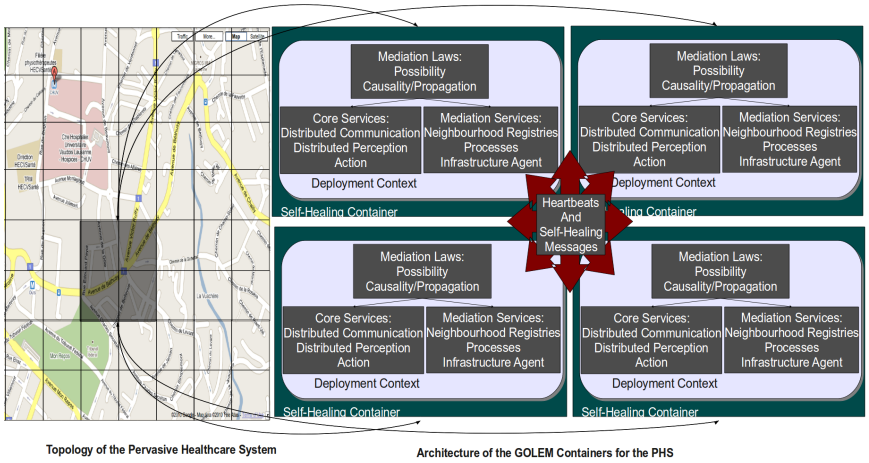

Fig. 1 Division in Containers of PHSs(on the left) and System Logic Architecture (on the right).

propagate the events produced by the infrastructure agents and to modify the agent environment topology when a disruption takes place. The infrastructure agents use the containers interfaces to perceive topological changes and cover for dead neighbours, while the rules take care of modifying the known neighbourhood by updating the neighbourhood registries. Every GOLEM container in this setting is connected with a neighbourhood and every container shares the same architecture with the neighbour containers, following the pattern of time-based redundancy. The rationale is a system that fulfils the requirements of self-healing systems: our system adapts to changes in the topology, it is observable as we use a declarative approach to describe our entities, it is autonomous and aware as we use planning agents to deal with the failures by coordinating in a distributed environment.

2.1 The Containers Behaviour

We specify the observable state of the GOLEM entities and of events as C-logic structures. C-logic [3] is a convenient formalism to express complex structures as logical objects, and it has a direct translation to the Ambient Event Calculus [2] (AEC), a formalism that handles the evolution in time of logical objects by means of events in distributed settings. In our GOLEM-based PHS, we express the state of a container by means of the following C-logic structure and AEC translation:

container:c1[position⇒{Latitude,Longitude},side⇒50,state⇒up,neighbours⇒{container:c2,container:c3}]

⇓

happens(ev1,0). instance(c1, container,start(ev1)). object(c1,side,50,start(ev1)).
object(c1,position, Latitude,start(ev1)). object(c1,position, Longitude,start(ev1)).
object(c1,state,up,start(ev1)). object(c1, neighbour, c2,start(ev1)). object(c1, neighbour, c3,start(ev1)).

which means that a container represents a real world location in terms of latitude Latitude and longitude Longitude, its state is up, it covers a square that has a side of 50 meters and it has two neighbours c2 and c3. GOLEM containers are programmed declaratively, using the AEC formalism. Through happens/2 predicates (the 2 represents the predicate arity) we specify how events take place in the AEC:

R1) happens(ack:Event [receiver ⇒ Container], T) ←
happens(Event[actor ⇒ Agent, receivers ⇒ Containers, known neighbours ⇒ NeighbourList], T),
member(Container, Containers), holds_at(Container, neighbour, this, T).

R2) happens(inform:Event[receiver⇒ Container], T) ←
happens(Event[actor ⇒ Agent, receivers⇒SubList, cover ⇒CBroken, randomvalue ⇒ Diceroll], T),
holds_at(this, neighbourhood_list, List, T), subset(List, SubList),
member(Container, SubList), holds_at(Container, neighbour, CBroken,T).

where the happens/2 is an AEC predicate stating that an event has happened in a container of the agent environment and holds_at/4 is an AEC predicate, that provides an attribute value given an entity identifier (in this case this represents the current container), the attribute name (in this case neighbourhood_list) and the time. R1 mediates an ack event produced by an agent during the PHS normal behaviour and it states that whenever such an event happens, then this also happens in the containers within the Containers list. The happens/2 has the function of replicating in the neighbourhood of a container an event happened locally through another happens/2. Inside the ack message, there is also the known neighbour containers list at a given time, so that every agent in the distributed topology can have knowledge of the neighbours of their direct neighbours. This is similar to the successor list of the CHORD P2P algorithm [10], where given a successors list of length r, and a disruption probability p for a single node, then the CHORD ring disruption probability is p^r, meaning that the ring resilience can be improved by increasing the successors list length. In our case, the probability that the PHSs cannot restore the area covered by a node is p^8, when keeping a list of neighbours of neighbours in a grid like topology, where if needed the resiliency of our PHS can be improved by increasing the neighbourhood knowledge.

R2 mediates disruptions happening in the distributed settings. Once an agent fails to send an ack events to a neighbour container that is down due to external causes, the agent sends an inform event to all the containers that have a neighbouring relationship with the unresponsive container and it starts the healing procedure that we will discuss later. Additional predicates have been defined to update the neighbours list when a communication fails and when a neighbour is substituted by another container. For the moment, the tasks of joining a network and redeploying a failed container are handled by a human actor. We will address these issues in future work.

2.2 The Infrastructure Agents

A GOLEM agent consists of a declarative module embedded in an agent body, which is situated in a container to perceive the events happening in it. The infrastructure agent cognitive model is based on two cycles, one to process the events sensed

by the body and one to plan and act in the environment. The pseudo code for the two agent mind cycles is reported here (CSP stands for Conditional-STRIPS-Planner, an extension of the STRIP planner [4] to handle conditional plans):

procedure ACTING-Cycle(time)
static:KB, a knowledge base; ACTION-QUEUE, a queue of actions accessible by the agent body;
$p_1, p_2 \ldots p_k, where \forall p_i, p_i \in Plans \subset KB$, a set of plans;
$currentstate \leftarrow$ STATE-DESCRIPTION(KB,time); $goal \leftarrow$ NEXT-GOAL($currentstate$,$time$);
if $\nexists p_i | p_i.goal = goal$ **then** $p_k \leftarrow$ CSP($currentstate$,$goal$); ADD(Plans, p_k);

$p_{exec} \leftarrow$ NEXT-EXECUTABLE-PLAN(Plans,$time$),

if($p_{exec} = nil$) **then** NOW(timenew); ACTING-Cycle(timenew);
else $currentact = p_{exec}.nextact$,
if(CONDITIONAL?($currentact$)) **then**
if(CHECK-KB(KB,IF-PART[$currentact$]))
then $p_{exec} \leftarrow$ THEN-PART[$currentact$]; ADD(Plans, p_{exec}); NOW($tnew$); ACTING-Cycle($tnew$);
else $p_{exec} \leftarrow$ ELSE-PART[$currentact$]; ADD(Plans, p_{exec}); NOW($tnew$); ACTING-Cycle($tnew$);
else ADD-ACTION(ACTION-QUEUE, $currentaction$); NOW($tnew$); ACTING-Cycle($tnew$);

procedure PERCEPTION-Cycle($time$)
static:KB; PERCEPTION-QUEUE; $p_1, p_2 \ldots p_k where \forall p_i, p_i \in Plans \subset KB$;
$percept \leftarrow$ NEXT-PERCEPT(PERCEPTION-QUEUE, $time$);
UPDATE-KB(KB, $percept$, $time$); NOW($timenew$), PERCEPTION-Cycle($timenew$)

The PERCEPTION-Cycle/1 reads the PERCEPTION-QUEUE for percepts coming from the environment. Such percepts are used to update the knowledge base KB about the state of the containers that are known in the agent environment. A plan is represented in the agent mind as a C-logic object. For example, the following plan expressed in AEC:

```
plan:p1[goal ⇒ cover:g1[container⇒ c2], diceroll ⇒ 3000,next_action ⇒ ac1, delay_action ⇒ 0,
  sequence ⇒ {inform:ev4[cover ⇒ c2, diceroll ⇒ 3000],wait:ev5[delay⇒ 6], if_then_else:if1 }].

if_then_else:if1[if ⇒ check_winner(ag1,p1,3000),
                 then ⇒ { modify_topology:ev6[cover⇒ c2], end_plan:ev8 }, else ⇒ { end_plan:ev9 } ]

check_winner(A,P, Diceroll)← now(Time),
     not (holds_at(P,competitor, Comp,Time),holds_at(Comp, diceroll, Diceroll*,Time), Diceroll* >Diceroll).
```

specifies a plan p1, with a goal g1 to cover for a container c2. In this plan the actions performed are: an inform event ev4, then a wait ev5 action with a 6 seconds delay, to attend for messages incoming from other agents. These actions are pushed by the ACTING-Cycle/1 in the ACTION-QUEUE/2 of the agent body, that produces them in the container, which then mediates the actions according to the rules previously defined. The inform event is sent to the neighbourhood of the dead container. Inside this event there is a random value diceroll that is used by the agents to compete for the coverage of a container (in this case c2). After waiting for the delay, the planner finds an if1 object as the next action to perform. This is a C-logic object representing an if-then-else structure, handled by the CONDITIONAL?/1 predicate in the planner, that checks for the if condition check_winner/3 in the if-then-else structure. The planner then executes either the then sequence of actions or the else sequence of actions according to the result of the check_winner/3 condition. If agent ag1 is the competition winner, it covers for the dead container and ends the plan, otherwise the plan is destroyed. Furthermore the NEXT-EXECUTABLE-PLAN/2 predicate distinguishes between plans that have been frozen due to the

introduction of a delay, and plans whose execution can continue, allowing for plans with different goals to be executed in parallel. Despite the communication taking place between the agents, two or more containers may end up covering the same area. When the inconsistency is perceived, the interested agents start a resolution protocol similar to the one to cover for a dead neighbour. High level planning agents have the advantage of a compact definition of the agent mind as plans are structures that can be instantiated and destroyed according to the interaction state, while reactive agents would require a low level verbose agent mind, that is difficult to debug in distributed settings.

3 Evaluation and Related Work

To evaluate our system we deployed a 10x10 grid of self-healing GOLEM containers in a dual core Intel Centrino 2, 2.66 Ghz with 3Gb of RAM.

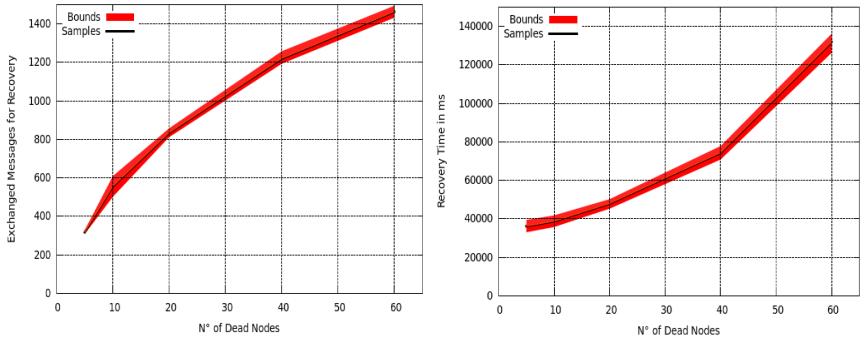

Fig. 2 Performance Evaluation of a 10x10 Pervasive Agent Environment

A PHS should be distributed on a grid and deploying it on a single host gives only a partial view of the real performances of the system. In real settings we foresee that some of the containers will require more resources when representing hot spots (i.e. super markets or hospitals), while unpopulated areas will require less resources. This is a matter of future work, but we recognise the need to study the system in real settings. Still, this evaluation offers important insights about the system performances. As evaluated by Urovi et al. in [11], a GOLEM container can support up to 50 users, meaning that a 100 containers grid, as in this evaluation, supports up to 4000-5000 users. This is a meaningful scale for a real system deployed in medium sized cities where the number of patients needing constant monitoring is on the order of hundreds. We took an average of 10 samples for each of the two curves in Fig. 2, that evaluate respectively the number of messages to recover from failures with an increasing number of containers and the total downtime with respect to the number of dead containers. These are two critical parameters: if too many messages

are exchanged this can impact the PHS performances, and if there are long downtimes, emergencies may happen when the system is unavailable. When producing the curves in Fig. 2 we made these assumptions: a) since the system is deployed in a single host, the delays of a real network are not taken into consideration b) we assumed that the containers die all at the same time, that is a pessimistic assumption, as the probability of this happening is very low; c) the system presents failures after every container had time to learn about its neighbours.

The first part of Fig. 2 has a logarithmic behaviour because the more nodes in a neighbourhood die, the less nodes take part to the competition for covering a dead node, producing less messages. Consequently, the second part of Fig. 2 behaves like a quadratic curve as the alive infrastructure agents have to execute more plans to cover a bigger area. Finally, the introduction of parallel plans helps agents to minimise the downtime as whenever a dead neighbour is detected, a new plan is instantiated to cover it. Also, introducing agent communication, mediated by the agent environment, allows to minimise a) the number of messages exchanged in the environment b) the uncontrolled growth of the area controlled by a container and c) the conflicts arising in the healing protocol.

Self-healing is a important topic within the distributed system community. Mikic-Rakic et al. in [9] propose the PRISM model for self-healing and fault tolerance. Such a model has a set of components connected by means of communication ports that exchange synchronous and asynchronous events and that rely on metal-level components for the self-healing procedures. With respect to PRISM, the infrastructure agents are meta-level components communicating and sharing knowledge about the topology of the distributed system. From the stand point of self-healing systems, in [6] Selvin et al. and Kondacs in [8], propose to utilise a bio-inspired approach to rebuild geometric shapes. These works demonstrate that using nature inspired models allows to have a very resilient service capable of self-healing 99% of its components. Our approach is similar to the one proposed in [8] and [6] except that we have further constraints on the number of messages exchanged and on the recovery time. Another contribution that uses agents is the one of Haesevoets et al. in [13], where the MACODO system defines laws in the agent environment to handle the consistency of the agent roles. As in [13], we separate the concerns of handling self-adaptation between the agent environment and the agents by means of cognitive agents dealing with the changes of the environment.

4 Conclusion and Future Works

In this paper we presented a pervasive healthcare system where agents reorganise the agent environment to self-heal from a fault of one or more of the containers composing it. We utilise planning agents that can reason in parallel about multiple faults and that produce plans and interact to cover for missing containers in the environment. The novelty of the approach resides in using planning agents combined with a complex declarative agent environment that simplifies the interaction between the

agents controlling the distributed system. We evaluated the system downtime and the number of message exchanged to recover from a growing number of dead containers, discovering that the system scales up and it can recover in useful time even when more than 50% of the system is down. Future works include deploying the system in real setting and testing it with real users as well as extending the algorithm to define the topology of the environment dynamically at deployment time. Another interesting issue for future work is how to deal with patients roaming in a distributed network that is self-healing from a disruption.

References

1. S. Bromuri, M. I. Schumacher, and K. Stathis. Towards distributed agent environments for pervasive healthcare. In *Proceedings of the Eighth German Conference on Multi Agents System Technologies (MATES '10)*, 2010.
2. S. Bromuri and K. Stathis. Distributed Agent Environments in the Ambient Event Calculus. In *DEBS '09: Proceedings of the third international conference on Distributed event-based systems*, New York, NY, USA, 2009. ACM.
3. W. Chen and D. S. Warren. C-logic of Complex Objects. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 369–378, New York, NY, USA, 1989. ACM Press.
4. R. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *IJCAI*, pages 608–620, 1971.
5. F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31:1–26, March 1999.
6. S. George, D. Evans, and L. Davidson. A biologically inspired programming model for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 102–104, New York, NY, USA, 2002. ACM.
7. D. Ghosh, R. Sharman, H. R. Rao, and S. Upadhyaya. Self-healing systems – survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007. Decision Support Systems in Emerging Economies.
8. A. Kondacs. Biologically-inspired self-assembly of two-dimensional shapes using global-to-local compilation. In *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*, pages 633–638, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.
9. M. Mikic-Rakic, N. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 49–54, 2002.
10. I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11:17–32, February 2003.
11. V. Urovi, S. Bromuri, K. Stathis, and A. Artikis. Towards runtime support for norm-governed multi-agent systems. In F. Lin, U. Sattler, and M. Truszczynski, editors, *KR*. AAAI Press, 2010.
12. U. Varshney. *Pervasive Healthcare Computing: EMR/EHR, Wireless and Health Monitoring*. Springer Publishing Company, Incorporated, 2009.
13. D. Weyns, R. Haesevoets, A. Helleboogh, T. Holvoet, and W. Joosen. The macodo middleware for context-driven dynamic agent organzations. *ACM Transactions on Autonomous and Adaptive Systems*, 5(1):3.1–3.29, February 2010.
14. D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.
15. M. Wooldridge. *MultiAgent Systems*. John Wiley and Sons, 2002.