Embryonic Models for Self–healing Distributed Services*

Daniele Miorandi[1], David Lowe[1,2], and Lidia Yamamoto[3]

[1] CREATE-NET, v. alla Cascata 56/C, 38100 – Povo, Trento, IT
daniele.miorandi@create-net.org
[2] Centre for Real-Time Information Networks, University of Technology, Sydney
PO Box 123, Broadway 2007 NSW, Australia
david.lowe@uts.edu.au
[3] Computer Science Department, Bernoullistrasse 16, CH - 4056 Basel, Switzerland
Lidia.Yamamoto@unibas.ch

Abstract. A major research challenge in distributed systems is the design of services that incorporate robustness to events such as network changes and node faults. In this paper we describe an approach – which we refer to as *EmbryoWare* – that is inspired by cellular development and differentiation processes. The approach uses "artificial stem cells" in the form of totipotent nodes that differentiate into the different types needed to obtain the desired system–level behaviour. Each node has a genome that contains the full service specification, as well as rules for the differentiation process. We describe the system architecture and present simulation results that assess the overall performance and fault tolerance properties of the system in a decentralized network monitoring scenario.

Keywords: distributed services, autonomic computing, self–healing behaviour, robustness, embryogenesis, differentiation mechanisms.

1 Introduction

In this paper, we address the problem of devising architectures and methods for robust and self–healing distributed services. Given a service whose execution involves tasks running on a plurality of interconnected machines (or nodes), we introduce techniques for coping with faults and ensuring robustness at the system level.

The motivation for our work comes from the increasing utilisation of distributed services, i.e. services whose outcomes depend on the interaction of different components possibly running on different processors. Distributed services typically require complex design with regard to the distribution and coordination of the system components. They are also prone to errors related to possible

* This work has been partially supported by the European Commission within the framework of the BIONETS project EU-IST-FET-SAC-FP6-027748, www.bionets.eu

E. Altman et al. (Eds.): Bionetics 2009, LNICST 39, pp. 152–166, 2010.

faults in one (or more) of the nodes where the components execute. This is particularly significant for applications that reside on open, uncontrolled, rapidly evolving and large–scale environments, where the resources used for providing the service may not be on dedicated servers (as the case in many grid or cloud computing applications) but rather utilise spare resources, such as those present in user's desktops or even mobile devices. (Examples of such scenarios are the various projects making use of the BOINC or similar platforms[1].) Other examples of distributed applications where each node takes on specific functionality include: peer-to-peer file sharing; distributed databases and network file systems; distributed simulation engines and multiplayer games; pervasive computing [13] and amorphous computing [1]. With all of these applications there is a clear need to employ mechanisms ensuring the system's ability to detect faults and recover automatically, restoring system–level functionalities in the shortest possible time.

In this paper we discuss an approach to addressing these issues that is inspired by cellular development and differentiation processes. Similar techniques have been applied in the evolvable hardware domain, giving rise to a specific research field called *embryonics* [10,11]. We propose an approach that utilises distributed nodes capable of differentiating into various types based on local knowledge, and which collectively lead to the emergence of the desired behaviour.

2 Background and Related Work

Robustness and reliability in distributed computing systems are well–studied topics. Classical fault–tolerance techniques include the use of redundancy (letting multiple nodes perform the same job) and/or the definition of a set of rules triggering a system reconfiguration after a fault has been detected [3]. When the scale of the system grows large, however, it is not practically feasible to pre–engineer in the system's blueprint all possible failure patterns and the consequent actions to be taken for restoring global functionalities. Such an issue is reminiscent of the reasons that led to the launch, by IBM, of the Autonomic Computing initiative [5], according to which one of the desirable properties of an autonomic system is *self–healing*. A self–healing system must recover full functionality, "healing" itself from faults and defects by actually fixing them autonomously, instead of just bypassing them.

In previous work by two of the authors [8,9], we considered the potential for using bottom-up approaches inspired by embryology to the automated creation and evolution of software. In these approaches, complexity emerges from interactions among simpler units. It was argued that this emergent behaviour can also inherently introduce self–healing as one of the constituent properties.

Our approach described in this paper builds on these concepts, leveraging off previous research conducted in the evolvable hardware domain on the application of architectures and methods inspired by the cellular developmental

[1] `http://boinc.berkeley.edu/`

and differentiation processes. Such approaches, which gave rise to the *embryonics* field [10,15], are based on the use of "artificial stem cells" [7,11], in the form of totipotent entities that can differentiate —upon reception of relevant signalling from nearby cells— into any component needed to obtain the desired system–level behaviour. Such an architecture has been successfully applied to field programmable gate arrays (FPGAs), resulting in the design of robust (i.e., able to sustain a large number of failures) and self–healing (i.e., able to recover automatically from faults by re-arranging its internal structures) hardware systems [10,14]. However, this earlier work did not consider the application of these approaches to distributed software applications.

In our work, we apply the concepts and tools developed in embryonics to the domain of *distributed software systems*. Such an application requires rethinking some of the design choices made by the embryonics research community to adapt to the specific features and constraints arising when working with software that is distributed over a network of interconnected machines. For example, network characteristics such as latency and dropped data packets can become a more significant factor in affecting the performance of the system. We call the resulting systems *EmbryoWare* (i.e. Embryonic Software).

Our approach bears many similarities to the work of Magrath [6], insofar as cells can propagate through a network and interact to achieve a global goal. The goal in Magrath's work is however to achieve a global pattern of cells (described as a phenotype) rather than a global behaviour. The cells in Magrath's approach also have a fixed set of behaviours that are not dependant upon the cell type (i.e. they do not change type). Further, their behaviour is dependant upon the network configuration of the neighbourhood rather than the cell types in the neighbourhood. This constrains the overall patterns of behaviour that can emerge from the network. The work described by Magrath does, however raise interesting questions with regard to how the cell genome can be designed so as to achieve a particular global pattern (or phenotype) – a question that is also relevant in our work.

Related approaches have also been recently proposed for enabling autonomic load–balancing among different application servers in a network [12]. Similar work by Chanprasert and Suzuki [2] considered the issue of self–healing in complex networks. Whilst not based on embryonics, the approaches were nevertheless bio-inspired (at the level of interacting individuals, rather than specialising cells), and demonstrated how distributed or decentralized control and processes of natural selection can lead to robust solutions.

3 Embryoware: Embryonic Software

EmbryoWare takes inspiration from the embryological or developmental processes in biology, by which an embryo made of initially identical cells (*stem cells*) develops into a full organism in which every cell assumes a different, specialized function, e.g. blood cells, skin cells, neurons. Stem cells are *totipotent*, i.e. unspecific and able to differentiate into the various cell types needed.

In EmbryoWare, *software stem cells* contain a genome with a concise representation of the *complete* service process to be performed. Such artificial stem cells are initially totipotent and are designed to spread throughout the network by self-replication. These cells differentiate into the various components needed for performing the overall service. Adequate signalling mechanisms shall be provisioned, so that cells could exchange information about the state of their neighbours. Upon detection of a fault in a neighbouring cell, they are able to re-enter the embryo state (unlike in biology), for differentiating again into the required functionalities, expressing the necessary genes.

It is important to note that, like most bio-inspired approaches, EmbryoWare remains an analogy, and is not meant to be entirely faithful to biology. A number of notable differences from true embryological processes can be highlighted. For example, unlike most biological examples, the EmbryoWare cells retain totipotency throughout their life, and are always able to re-differentiate into other cell types as needed. This is also done in other embryology-inspired approaches such as embryonics [10,15]. Similarly, real embryo formation makes use of on apoptosis (i.e. programmed cell death), whereas in our framework nodes will continue operation indefinitely (or until the operational task is completed).

3.1 Architecture and Components

The EmbryoWare approach is based on the use of nodes or cells (understood as basic computational units) [2] possessing the ability to decide autonomously, based on the task currently performed (referred to as *"cell type"* in the following) and on the ones performed by nearby cells, which task should be performed next in order to maximize the benefit for the system as a whole. Each cell is provided with a complete system–level specification of the service to be performed, called the *genome*, which includes:

(i) a description of the expected behaviour of the service as a whole;
(ii) a description of the single tasks to be performed by cells;
(iii) a set of rules for deciding, based on the current task and the task performed by neighbouring cells, which task is to be performed next.

Each genome comprises a finite number of functions to be executed. We say that a cell performing a given function has *differentiated* into a given *type*. The type of a cell defines therefore its current role in the system–level architecture. A cell containing the genome can differentiate into any specific cell type encompassed by the genome.

Cells are arranged in a graph topology. The immediate neighbours (or 1-hop neighbors) of a cell are defined as the cells that are able to communicate directly with it. The n-hop neighbourhood consists of nodes located n communication hops away. In the embryonics domain, cells are often arranged in a toroidal regular grid, akin to a *cellular automaton (CA)*, where each cell is connected only

[2] Throughout the paper, we use the words *cell* and *node* interchangeably, as well as the words *task* and *function*.

with its immediate four (von Neumann neighborhood) or eight (Moore neighborhood) neighbours. Toroidal grids avoid border effects, but are rarely found in practical network scenarios (e.g. sensors spread over a field to be monitored) where border effects cannot be neglected. In our work we have not assumed this for the general case, and the algorithms are independant of the network topology.

An EmbryoWare system consists of the following components:

- *Genome*: defines the behaviour of the service as a whole, and determines the type to be expressed based on the local context (i.e., neighbouring cell types).
- *Sensing agent*: software component that periodically communicates with neighbours regarding their current type. The type of a cell is maintained in a separate register;
- *Replication agent*: software component that periodically polls the neighbours about the presence of a genome; if a genome is not present then the current genome is copied to the "empty" cell;
- *Differentiation agent*: software component that periodically decides, based on the cell's current type and the knowledge about the types of the neighbouring cells, which functions should be performed by the node.

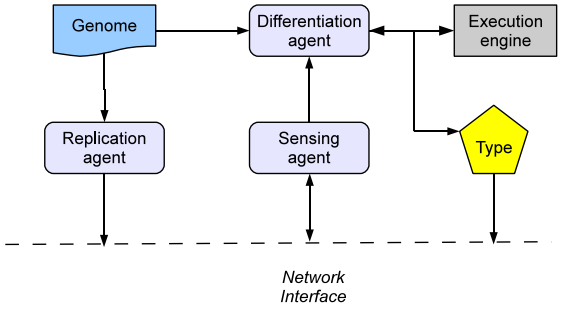

Fig. 1. Architecture of EmbryoWare: single–node view

A possible node–level architecture for EmbryoWare is shown in Fig. 1. The genome is connected to both the replication agent (which tries to replicate it in neighbouring "empty" cells) and to the differentiation agent. The latter also receives information from the sensing agent on the status of neighbouring cells. The cell's "type" is maintained in a separate register. It is also communicated to the execution engine, which performs the tasks/actions associated to the current type. The outcome of the execution process may trigger a differentiation (as in the case in which, e.g., the execution cannot be performed successfully due to some faults in the genome). The type of a cell can be read by the sensing agent of a neighboring node. The differentiation process can be implemented in a variety of ways. The simplest one is a lookup table (similar to those used in CAs), that

determines, based on the current type of a node and on the sensed type of its neighbours, which type it should differentiate into.

In order to limit potential security issues, we further add *autonomy* constraints. Namely, a cell cannot be "reprogrammed" by a peer, but it will decide autonomously on the function to be performed (taking into proper account the local context). Hence a cell can influence only in an indirect way (by setting its own personal 'type') what neighbouring cells will do. Such a feature is appealing in that it limits the possibility of a malicious host affecting the emerging system–level behaviour.[3]

The two key aspects of EmbryoWare are related to (i) the development of an adequate representation of the service as a whole, able to be at the same time concise and expressive (ii) the development of efficient techniques for handling the differentiation process, requiring only local information from surrounding nodes. It is important to remark that the EmbryoWare architecture fits well services where the role of a given cell depends only on the type of neighbouring cells. While appropriate signalling mechanisms can be put in place to exchange information among remote nodes (enlarging in such a way the possible application domain), the resulting overhead may limit the system's performance. At the same time, communications among remote cells can be needed to obtain the desired system–level functionalities. Such a feature is supported by the system. What in EmbryoWare shall be limited is the signalling needed between cells in order to perform differentiation. We illustrate this issue in the case study presented later in the paper.

4 Algorithms for Embryonic Software

Given the architecture presented in the previous section, we may identify three key operations to be performed within an EmbryoWare–type system: *sensing*, *differentiation* and *replication*. In this section, we present algorithms for performing such functions in a distributed and asynchronous way.

4.1 Sensing Process

The sensing process is performed periodically at each node. Every τ_1 seconds, the cell issues a `queryType` message to its 1-hop neighbours, which reply sending information about their current type. The list of neighbours (indicated as `NeighboursList`) is created at bootstrap; its setup and maintenance is deferred to appropriate network–level services and is therefore not described in this work.

[3] It is however worth noticing that such an approach does not prevent malicious hosts from influencing the behaviour of the system. Proper security countermeasures should be put in place to limit the possible impact of such an occurrence. Further, it is worth remarking that the replication of genome in nearby cells require, in order to avoid potentially disruptive interference by malicious nodes, to put in place appropriate authorization and authentication procedures.

The information about the type of neighbours, `Type(·)`, maintained in an appropriate knowledge base, is then updated. If a cell is faulty (i.e., machine is down due to maintenance or technical problems), it will not reply to the `queryType` message. Every node maintains therefore a timer, associated to a timeout, for each query message sent. In the case where no reply is obtained from a neighbour within the given timeout, its type is set to 'faulty'. The sensing process can be executed serially (polling one neighbour at a time) or in parallel (sending out queries to all neighbours and waiting for the message replies). Alg. 1 details the algorithm for the parallel case.

```
loop
   every τ1
   for all i ∈ NeighbourList do
      send queryType message to i
      instantiate timer(i) for node i
   if timer(i) expires then
      Type(i) ← FAULTY
   if received message from i then
      update Type(i)
```

Algorithm 1. Sensing algorithm pseudo-code (parallel)

4.2 Differentiation Process

As with sensing, the differentiation process is performed periodically at each cell. We denote by τ_2 the differentiation period. Cells are not necessarily synchronized, so that the differentiation process can take place at different time instants at different nodes. In general, there is no need to specify a particular coupling between the differentiation period and the sensing period (they can well be implemented as independent threads). However, to reduce redundant processing, the period of the cell differentiation process should be equal to or larger than the sensing period τ_1.

The differentiation process is represented as a set of rules (which may be coded as a lookup table) provided as part of the genome.[4] Each cell uses information about its current type and the type of neighbouring cells to decide which type to express next (i.e., which function to be performed). The mechanism can be deterministic (given current state x and neighbours $1, \ldots, k$ in state $y_1, \ldots, y_k$, move to state z) or probabilistic (given current state x and neighbours $1, \ldots, k$ in state $y_1, \ldots, y_k$, move to state z_1 with probability p_1, to state z_2 with probability p_2 etc). A possible implementation of the differentiation algorithm is shown in Alg. 2.

[4] In general, other methods can be envisioned, based on, e.g., reaction–diffusion patterns [4]. It is also possible to envision accounting for environmental variables (such as, e.g., current CPU load or other contextual information) in the differentiation process. In this work, we limit our attention to a simpler set of rules only for the sake of simplicity.

```
loop
    every τ2
    read Type(myID)
    for all i ∈ NeighbourList do
        read Type(i) {Update information on neighbour's type.}
    LOOKUP < Type(myID), Type(i1), ..., Type(ik)) >
    update Type(myID)
```

Algorithm 2. Differentiation algorithm pseudo-code

4.3 Replication Process

The replication process is meant to ensure that the system can make use of spare resources (empty cells that have not yet had a genome inserted into them by a neighbouring cell) whenever available, and hence the ability (at the system level) to recover from major faults. It could also be seen as a mechanism for automating service deployment in a distributed system. Through a suitable replication process, it would be sufficient to inject a "seed" genome into the system, and it will replicate itself across the network and differentiate into the necessary components. We assume that only the genome is replicated onto empty nodes: the management components (differentiation agent, replication agent, sensing agent, execution engine) are assumed to be present on all nodes in the system as part of the basic node platform. The replication algorithm works by periodically inquiring all neighbour cells about the presence of a genome. A node without an installed genome is still able to respond to queries, but will indicate that it has no functioning genome. If the cell is found to be empty, a copy of the genome is transmitted to the empty node, where it is installed and initiated. A possible implementation of the replication process is described in Alg. 3. While the replication period τ_3 is not strictly related to the sensing and differentiation period, the following relation provides an ordering suitable to maintain a good level of performance: $\tau_3 \gg \tau_2 \geq \tau_1$.

```
loop
    every τ3
    for all i ∈ NeighbourList do
        send isGenomePresent message to i
        if noGenomePresent message received then
            send Genome to i
```

Algorithm 3. Replication algorithm pseudo-code

5 Evaluation on a Decentralized Monitoring Scenario

We evaluate the proposed techniques with a case study in the domain of decentralized network monitoring. Consider a sensor network, or any large network of devices where environment or system parameters must be monitored, and alarms must be raised whenever abnormal circumstances are detected. In such a scenario, nodes may perform different tasks, e.g. sense, collect, filter and log information, and then finally decide whether an alarm should be raised or not,

based on the information sensed. This is a typical scenario where the automatic differentiation into each of these separate tasks, performed by EmbryoWare, is a helpful feature in order to keep providing a prompt and reliable service in spite of node failures or unexpected network changes.

5.1 Case Study Description

We consider a network of cells performing resource monitoring, logging, and alarm generation. We assume that each cell possesses some parameter that needs to be periodically monitored. Each *monitoring* sample needs to be reported to a *logging* cell that should be no more than 2 hops away from the *monitoring* cell in order minimise network traffic[5]. The *logging* cell will accumulate data samples and report them periodically to *alarm* cells. The network should contain 2 *alarm* cells for redundancy, but no more than 2 *alarm* cells in order to minimise the resource requirements associated with alarms. Each *alarm* cell should have two 1–hop neighbours that are *analysis* cells which it uses to assist in analysing the provided samples. When the alarm cell recognises an alarm condition on one of the data samples this is reported in a system-dependant fashion.

In addition, in order to distribute the load associated with alarm condition evaluation, the cells taking on the *alarm* behaviour should change periodically. Further, to evaluate fault performance, our simulation includes random genome failures (with a predefined probability that is independent in our case study of the cell type). A genome fault is where the genome can no longer operate correctly – and hence will not respond with a valid genome type when queried by a neighbouring node. The underlying management components are however still operational, and so a neighbour could reinsert a new genome to correct the genome fault. Conversely, a node failure is where the node becomes permanently inoperable.

5.2 Cell Types and Their Behavior

The desired behaviour requires a genome with the following types: Stem, Faulty, Monitor, Logger, Analysis, Alarm. A *stem* cell in our system is one that is currently idle but ready to differentiate into some needed type. The specific behaviours outlined above can then be obtained in a number of different ways, with different implications for the cell type patterns that emerge, and the timing within which the differentiation happens.

A crucial aspect is how to maintain a global target number of alarm nodes ($N = 2$ in the case study), in a decentralized way. Presently, this is achieved by letting nodes broadcast a beacon when they become alarm ones. Each cell maintains a list of active alarm nodes. When any logging cell is ready to send its data, it will try to send to each alarm cell that is in its list, and if it does not receive an acknowledgment then it removes that alarm from its list.

[5] A 2-hop neighbourhood can be monitored by asking 1-hop neighbours about their neighbours. i.e. when a 1-hop neighbour reports its node type, in also includes relevant information about it's own 1-hop neighbours.

One key choice is with regard to the processes associated with differentiation. It is possible to implement the conversion of a cell's genome into an *alarm* type either proactively or reactively. In the pro-active case, each cell (through its sensing agent) will monitor the state of other cells and if it determines that there are not two *alarm* cells, then it can proactively differentiate into an *alarm* cell (albeit with a level of randomness to ensure that not all nodes differentiate into an *alarm* cell simultaneously). In the reactive case, when a *logging* node's execution engine attempts to transmit data to the *alarm* cells, if it receives no response then it can reactively trigger the differentiation into an *alarm* cell. Both variations have been implemented in order to compare the performance of the two approaches. The detailed implementation description is deferred to App. A.

5.3 Results and Assessment

The case study has been implemented and simulated with Matlab, using a regular Moore-neighbourhood grid. It illustrates the viability of both the general approach and the specific architecture that has been proposed. It also uncovers a number of performance considerations and system design guidelines.

As a first performance metric, we considered the fraction of time the system was in the 'up' state (i.e., a state in which the requirements in terms of presence of *loggers*, *alarms* and *analysis* nodes were met) as a function of the failure rate of single genomes. We considered a 10×10 network, *alarms* differentiating back to *stem* cells after 20 s, *stem* cells becoming *monitors* with probability (per second) of 0.1, *loggers* differentiating back to *stem* cells with probability (per second) of 0.001, *alarms* differentiating to *stem* cells with probability (per second) of 0.1, *alarms* decaying to *stem* cells with probability of 0.3 in case too many *alarms* are present in the system. At the beginning of the simulation, one single genome is injected into the node at position $(1, 2)$, and is left to replicate and differentiate in the system. For each value of the genome fault rate, 20 runs were performed, each one consisting of 3000 iterations of the differentiation process over the whole system. The genome fault rate (in s^{-1}) was varied between 10^{-3} to 0.9. The results obtained are plotted in Fig. 2 on a semi–logarithmic scale. As it can be seen, the fraction of time spent in an invalid state is only marginally sensitive to the genome fault probability, an appealing feature for system's designers. The presence of a "floor" on the system downtime depends on the high level of randomness present in the system; in particular the decay of nodes into the *stem* cell state leads to a non–zero probability of being in an invalid state even in the absence of genome faults.

As a second performance metric, we considered the time necessary for the system to reach a valid state (i.e., able to meet the requirements in terms of presence of *loggers*, *alarms* and *analysis* nodes) starting from a single genome injected into the cell $(2, 1)$. Such a parameters provide a measure of the time needed for a newly deployed system to settle into a valid state. We varied the network size between 36 and 400 nodes and used a genome failure rate of 0.01 s^{-1}. For each network size considered, 10 independent runs were executed. The other parameters were set as described above. The results obtained are plotted in

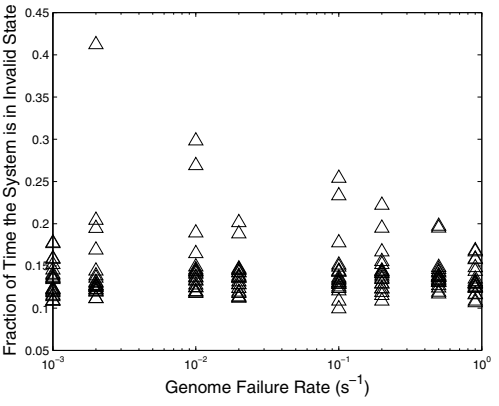

Fig. 2. Fraction of time the system is in an invalid configuration as a function of the genome fault rate

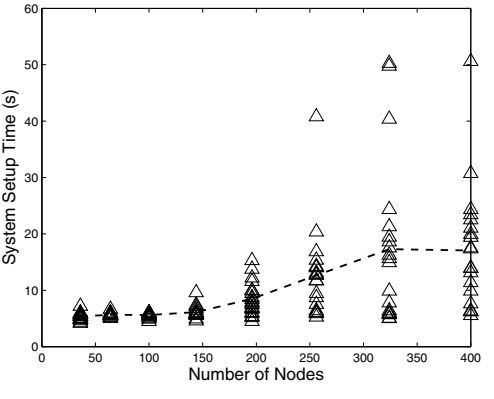

Fig. 3. Time necessary for reaching a valid configuration starting with a single genome injected in node $(2, 1)$ as a function of the network size

Fig. 3. The dashed line represents the mean values, while the outcomes of single runs are reported using triangular markers. As it can be seen, the time to reach a valid state increase with the network size, growing from ~ 5 s to ~ 15 s (in terms of mean values). Such an increase cannot be ascribed to communication delays, but is related to the fact that the probabilistic differentiation processes at the hearth of the example shown require careful tuning of parameters to offer good performance for different network size. In other words, the parameters driving the transitions between different types should be tuned according to the network size in order to achieve optimal performance. At the same time, while an "aggressive" behaviour (with rather high values for differentiation probability, and hence nodes volunteering more rapidly) can lead to a speed–up of the time taken to reach a valid state, it may also lead to undesirable oscillation during

normal operations. This is because too many nodes can differentiate into a particular role (e.g. the alarm in our scenario). When each node realises that there are too many volunteers, they then "aggressively" revert back - though again too many do so, setting up a cycle. This is a combination of high probabilities of conversion, coupled with the communication delays, meaning that cells make a differentiation decision before they have data on the fact that they have neighbours who have also done so. We are yet to investigate the circumstances under which such an oscillatory beahviour may occur.

Overall, the simulation demonstrated that robust fault tolerance, in the event of both node and genome faults, can be supported quite elegantly. In the case of genome faults, provided the node can detect a fault in its genome, it can purge it and allow the replication process to reinsert a new (operational) one. When a node itself fails then it will remain inoperable, but the simulation demonstrated that its functionality is subsequently accommodated by other ones.

The simulation also highlighted that in the current architecture the optimal tuning of the differentiation parameters is dependant upon the network size. For example, when an *alarm* differentiates back to a stem cell, *logger* nodes will probabilistically differentiate to form new *alarm* nodes. As the network size increases, the number of differentiations that occur, for the same probability parameters, will increase. This behavioural dependence upon network size is undesired – ideally the performance should be independent of the number of nodes. [6]

A number of genome design issues also emerged from the simulation. It became evident that network fringe effects need to be considered in designing the genome behaviours. For example, with a poorly designed genome, it is possible that an *alarm* node might appear on the border of the network where the requirement for two neighbouring *analysis* nodes cannot be met. This issue could be addressed in several ways.Nevertheless, this does highlight that careful consideration must be given to the genome design lest unintended behaviours emerge.

Another interesting behaviour that became apparent was what we might call the *Hydra*[7] behaviour. If we split the network of nodes into two isolated sub-networks, then the nodes differentiate in order to create a fully operational system in each sub-network, exhibiting a natural self-healing ability.

6 Conclusion

In this paper we have proposed *EmbryoWare*, an embryonic–inspired architecture for robust and self–healing distributed software. The approach is based on

[6] Such a dependence on the network size comes from the fact that the case study has to satisfy a *global* constraint (i.e., on the number of *alarm* nodes in the system). If only *local* constrains were present (e.g., one alarm cell shall be present within k hops from any loggers), the dependence on the network size would blur.

[7] The Hydra is a small freshwater animal that exhibits an interesting behaviour. If it is severed into multiple parts, then each part is capable of morphollaxis – i.e. reorganising / regenerating to become a fully functioning individual hydra.

each node in the system containing a genome that includes a complete specification of the service to be performed, as well as a set of rules that ensure each node differentiates into the node type required to provide required overall system behaviour. The simulations that we have performed have examined the case of genome failure and demonstrated the viability of this approach as well as the inherent robustness and self-healing that is achieved.

In ongoing work we will be considering other failure scenarios (e.g. link and node failures, changes in network topology, etc.) and how the system recovers from these failures. We will also be broadening the basis for analysing the system performance to include a more thorough analysis of how the tuning of the genome differentiation rules – and especially the stochastic parameters associated with decisions on the timing of the differentiation – affect the overall performance. We will also be evaluating the processing and communication overheads that this approach introduces, and how these scale with changes in the network size.

A number of additional research questions also emerge from these preliminary studies. Whilst our simulation captured relatively sophisticated behaviour, it was still less complex than many applications. It does raise the question of how complex a genome needs to be in order to provide desired behaviours, and whether a threshold will be reached where the genome complexity becomes prohibitive. Our evaluation also indicated the importance of considering carefully the processes required to understand and design for reliability and robustness – particularly in the context of network fringe effects. Subsequent work will also need to consider how to handle differences in the capabilities of nodes by adequately taking them into account in the differentiation process. The system sensitivity to various environmental characteristics, such as network size and network latency should also be considered.

Other future work could be to complement the EmbryoWare framework with an apoptosis or programmed cell death scheme, as a reverse operation for the current replication scheme. Apoptosis could be useful to optimize the placement of redundant functions (e.g. to minimize broadcast, etc.). It could also be used to deal with security breaches by isolating and killing misbehaving cells, a mechanism that is necessary when code can propagate in the network by replication. Apoptosis could also offer a mechanism by which, once the processing is complete in some regions of the network, nodes could "die" elegantly. Finally, another topic for future work would be to actually evolve the genome program to adapt to new situations.

References

1. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy Jr., G., Knight, T.F., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R.: Amorphous computing. Communications of the ACM 43(5), 74–82 (2000)
2. Champrasert, P., Suzuki, J.: A biologically-inspired autonomic architecture for self-healing data centers. In: 30th IEEE International Conference on Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 350–352. IEEE, Los Alamitos (2006)

3. Coulouris, G.F., Dollimore, J., Kindberg, T.: Distributed systems: concepts and design. Addison-Wesley Longman, Amsterdam (2005)
4. Deutsch, A., Dormann, S.: Cellular automaton modeling of biological pattern formation: characterization, applications, and analysis. Birkhäuser, Basel (2005)
5. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comp. Mag. 36(1), 41–50 (2003)
6. Magrath, S.: Morphogenic systems engineering for self-configuring networks, July 2-5 (2007)
7. Mange, D., Stauffer, A., Tempesti, G.: Embryonics: a microscopic view of the molecular architecture. In: Sipper, M., Mange, D., Pérez-Uribe, A. (eds.) ICES 1998. LNCS, vol. 1478, pp. 185–195. Springer, Heidelberg (1998)
8. Miorandi, D., Yamamoto, L.: Evolutionary and embryogenic approaches to autonomic systems. In: Proc. of ValueTools (InterPerf Workshop), Athens, Greece, pp. 1–12 (2008)
9. Miorandi, D., Yamamoto, L., De Pellegrini, F.: A survey of evolutionary and embryogenic approaches to autonomic networking. Computer Networks (in press, 2009), doi:10.1016/j.comnet.2009.08.021
10. Ortega-Sanchez, C., Mange, D., Smith, S., Tyrrell, A.: Embryonics: a bio-inspired cellular architecture with fault-tolerant properties. Genetic Programming and Evolvable Machines 1, 187–215 (2000)
11. Prodan, L., Tempesti, G., Mange, D., Stauffer, A.: Embryonics: artificial stem cells. In: Proc. of ALife VIII, pp. 101–105 (2002)
12. Saffre, F., Shackleton, M.: "embryo": an autonomic co-operative service management framework. In: Artificial Life XI: Proc. 11th Int. Conf. Simulation and Synthesis of Living Systems, pp. 513–520. MIT Press, Cambridge (2008)
13. Saha, D., Mukherjee, A.: Pervasive computing: A paradigm for the 21st century. Computer 36(3), 25–31 (2003)
14. Stauffer, A., Mange, D., Tempesti, G., Teuscher, C.: A Self-Repairing and Self-Healing Electronic Watch: The BioWatch. In: Evolvable Systems: From Biology to Hardware. LNCS, vol. 2210, pp. 112–127. Springer, Heidelberg (2001)
15. Tempesti, G., Mange, D., Stauffer, A.: Bio-inspired computing architectures: the *embryionics* approach. In: Proc. of IEEE CAMP (2005)

A Detailed Use–Case Implementation Description

To illustrate a specific Genome pattern, we describe the execution and differentiation behaviours for the case where the genome responds reactively to failures to find *logger* and *alarm* nodes. Algorithm 4 describes the execution behaviours. As can be seen, a *monitor* node can reactively trigger a differentiation into a *logger* node when required, and a *logger* node can reactively trigger a differentiation into an *alarm* node when required. Algorithm 5 shows both the proactive and reactive differentiation behaviours for the genome. The proactive differentiation is triggered by the relevant sensing of the node neighbourhood, whereas the reactive differentiation is triggered by events occurring in the execution engine.

- *Stem* cell:
  None
- *Faulty* cell:
  None
- *Monitor* cell:
  every T_M generate sample S
  repeat
  transmit S to *logger*
  if transmission failed **then**
  search 2-hop neighbourhood for logger
  if no *logger* found **then**
  trigger reactive differentiation
  until S processed
- *Logger* cell:
  accept, record all *Monitor* samples
  accept, register all *Alarm* beacons
  every T_L
  while recorded samples still to be transmitted **do**
  attempt transmit samples to all *alarms*
  for all *alarm* cells that do not respond **do**
  deregister *alarm*
  if $<$ 2 registered *alarms* **then**
  trigger reactive differentiation.
- *Analysis* cell:
  accept, process *Alarm* requests
- *Alarm* cell:
  accept, process *Logged* data
  send *Alarm* requests to analysis cells

Algorithm 4. Execution behaviours for typical Genome for data logging application.

- *Stem* cell:
  Proactive: with probability $P_{TtoM} \Rightarrow Type \leftarrow Monitor$
- *Faulty* cell:
  None
- *Monitor* cell:
  Reactive: no logger $\Rightarrow Type \leftarrow Logger$
  Proactive: *alarm* neighbour has $<$ 2 *analysis* cells $\wedge$ offer accepted $\Rightarrow Type \leftarrow analysis$
  Proactive: probability $P_{MtoT} \Rightarrow Type \leftarrow Stem$
- *Logger* cell:
  Reactive: if $<$ 2 alarms found, with probability $P_{LtoA} \Rightarrow Type \leftarrow Alarm$, broadcast beacon
  Proactive: with probability $P_{LtoT} \Rightarrow Type \leftarrow Stem$
- *Analysis* cell:
  Proactive: *alarm* not responding $\Rightarrow Type \leftarrow Stem$
- *Alarm* cell:
  Proactive: $>$ 2 registered *alarms* $\wedge$ probability $P_{MAtoT} \Rightarrow Type \leftarrow Stem$
  Proactive: active for $> T_{Alm} \wedge$ probability $P_{MtoT} \Rightarrow Type \leftarrow Stem$

Algorithm 5. Differentiation behaviours for typical Genome for data logging application.