FlashMob: Distributed Adaptive Self-Assembly

Daniel Sykes, Jeff Magee, Jeff Kramer
Department of Computing
Imperial College London
{daniel.sykes, j.magee, j.kramer}@imperial.ac.uk

ABSTRACT

Autonomous systems need to support dynamic software adaptation in order to handle the complexity and unpredictability of the execution environment, and the changing needs of the end user. Although a number of approaches have been proposed, few address a key issue: that of distribution.

In this paper we seek to overcome the limitations of centralised approaches. We build on our previous work on adaptive self-assembly within the three-layer model for autonomous systems to provide a decentralised technique for self-assembly. To achieve this in a fault-tolerant and scalable manner, we use a gossip protocol as a basis. While no central or leader node is aware of the full space of solutions, gossip ensures that agreement on a particular solution—in this case a component configuration—is reached in a logarithmic number of steps with respect to the size of the network.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures

General Terms
Management, Design, Reliability

Keywords
Self-adaptive, self-assembling, software architecture, distribution, gossip

1. INTRODUCTION

From smart phones, to unmanned drones and intelligent homes, distribution is a pervasive assumption in the modern computing environment. And yet, few adaptive systems have been proposed that specifically address this issue. One need only take a few examples to see the pattern: the repair strategies of dynamic Acme [6] and of Rainbow [3] assume a centralised unit which executes the strategy with a complete and consistent view of the architectural components of the system (even if those components are distributed); architectural planning [1, 17] requires a centralised planner; and the more abstract proposals including graph grammars [12], failure prediction [5] and model checking [20] assume a single model of the entire system, and speak nothing of decentralisation. The recent paper by Weyns et al. [19] also concludes that there is a dearth of decentralised approaches to self-adaptive systems.

Centralised systems are susceptible to well-known and well-studied problems encountered when no special attention is paid to distribution. On the one hand, the centralised unit (be it a model, a controller, a planner) can limit the performance of the system as a whole through having to process messages from every other node, having to perform computations on behalf of every node, and having to store information pertaining to every node. The computational and memory resources of the central unit can become exhausted.

On the other hand, distribution introduces questions of reliability both of the central unit and of the subservient nodes. Failure of the central unit leads to total failure of the system, just as if it comprised no distributed nodes at all. The one fault which a centralised adaptive system could never adapt to is the failure of the central controller.

Failure of a secondary node requires the central unit to take steps to ensure that the requirements of the system continue to be met, by engaging replacement nodes to take over the responsibilities of the failed node or by finding alternative ways to meet the requirements using fewer nodes. In other words, the system must adapt to overcome node failure.

Out of the many approaches for self-adaptive systems, there are few proposals—such as that by Georgiadis et al. [7]—which explicitly address the concerns of distribution and use node failure as an impetus for change. In that work, the central unit was, in effect, replicated so that each node maintained a model of the entire system along with its requirements. The consistency of the model was ensured by using reliable, totally-ordered broadcast, which limited the size of network the approach could support.

In this work, we propose an alternative means to deal with decentralisation in a component-based self-adaptive system. We overcome the scalability limitations found in the work of Georgiadis by making use of a gossip protocol [4] which guarantees that each node can find a consistent view of the whole system in a reasonable amount of time, without resorting to leader election and totally-ordered broadcast.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SEAMS’11, May 23–24, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0575-4/11/05 ...$10.00
This proposal is made in the setting of our overall approach for self-adaptive systems using the three-layer conceptual model [11] shown in Figure 1. This model divides the decision processes found in the plan step of the sense-plan-act feedback loop [2] into three layers, so that each kind of adaptation can be dealt with at the appropriate level of abstraction and in a timely manner.

Figure 1: Three-layer conceptual model

In the component layer, application behaviours are encapsulated in a configuration of components which are able to perform local domain-specific adaptations quickly. In the next layer, the change management layer, wider-ranging adaptations are enacted by manipulating the configuration of components in the layer below. Such adaptations naturally take longer, and are performed less frequently. The goal management layer provides the change management layer with a set of specific functional capabilities required to execute a plan. The plan is derived from abstract goals indicating the system requirements and a model of the environment. Adaptations in the goal management layer are the most time-consuming and infrequent in the system, but permit the entire behaviour of the system to be changed.

In previous work, the component control layer was (statistically) distributed, while the change management layer performed a centralised assembly and re-assembly process using declarative dependency, structural, and non-functional information to derive a configuration [15, 16]. In this work, we focus on decentralising the change management layer.

The rest of the paper is organised as follows. Section 2 describes the problem of adaptive self-assembly and how it was addressed in our previous work. Section 3 provides some background on gossip protocols before Section 4 describes our solution for distributed self-assembly using gossip. Section 5 gives some results from experiments using the technique, showing that configurations are agreed upon in logarithmic time. Section 7 concludes.

2. ADAPTIVE SELF-ASSEMBLY

The problem of adaptive self-assembly can be stated as follows: given a set of available components (with various functional and non-functional properties), and a configuration of components which are already running, find a new configuration which works (better) in the changed execution environment (including hardware), meets new user requirements or takes account of new component implementations.

In the context of the three-layer model, the user requirements come from the goal management layer, and the running components are found in the component control layer. Previously we proposed a centralised self-assembly process which took account of dependencies between components, structural constraints the user wished to enforce (such as an architectural style like pipe-and-filter), and annotations describing the non-functional (NF) properties of components, such as reliability, cost and performance [15, 16]. Assembly occurred initially to begin plan execution, and re-assembly occurred as necessary when goals changed or when components failed or their NF properties diverged from the claimed values.

A configuration produced using this procedure is called complete if it contains no unsatisfied requirement. A configuration is called valid if it additionally satisfies the structural constraints. Components are explicitly annotated with their provided and required interfaces\(^1\), and appropriate NF properties. A notion implicit in the previous work was that the components would be distributed among a set of physical hosts (nodes) connected in a network according to some secondary process, and that failure of a node hosting a component was identical with the failure of that component. Failure of the node deriving configurations would lead to total failure of the system.

To overcome this problem, the work described herein removes the assumption of a central node that is aware of the full set of available components (and hence the space of possible configurations). Instead, each peer node is aware only of the components it hosts locally, and the peers must reach agreement on a (global) configuration which has all the characteristics of a centrally-derived configuration: completeness, validity, and satisfaction of user requirements and NF preferences. Gossip is the means by which we achieve agreement.

3. GOSSIP FUNDAMENTALS

A gossip protocol provides an ideal means to achieve global agreement using decentralised information. Gossip protocols are used to ensure that a network of peers all receive some piece of information within a certain time, without resorting to reliable broadcast, which may involve a huge number of messages (restricting scalability). In our case the item of information is the (global) component configuration. There are two kinds of gossip protocol: the basic ones which involve a single update (which we shall call simple), and aggregate protocols, which, through the propagation of several updates, compute some aggregate function of information stored at each node. Our protocol is of the latter kind.

3.1 Simple Gossip

Gossip was originally inspired by the way in which disease epidemics spread across a population [4]. In its simplest form, gossip is a protocol for propagating a single piece of information (a database update for example [4]) across a network. Initially one node has the information (which we refer to as the state), and chooses another node at random to transmit the information to. The protocol proceeds in synchronous rounds so that in the next round, both the informed nodes independently choose another node to update. If all nodes continue transmitting indefinitely, the

\(^1\)These interfaces may use several modes of interaction, such as procedure calls, streams, events, and so on.
protocol is referred to as anti-entropy, and the information is guaranteed to reach all nodes in the network \[4\]. If nodes decide to stop transmitting after some number of rounds, the protocol is referred to as rumour mongering.

Variations of the anti-entropy protocol can be created by adjusting the strategy for selecting nodes (round robin is one alternative) or by performing extra steps when nodes communicate. With a push policy, the update is transmitted from the choosing node, while with a push-pull policy, updates pass in both directions. In the uniform push (anti-entropy) algorithm, a single state update will propagate across the network in \(\log_2 n + \ln n + O(1)\) rounds, where \(n\) is the size of the network \[4, 14\]. The speed of propagation ensures that gossip scales well: for 1000 nodes, dissemination of a single update may take as few as 17 rounds.

The random pattern of message exchange ensures that the loss of a single message only delays agreement since the node that would have received the update will eventually receive another message. Likewise if a node fails and restarts, it only has to wait for a single message to arrive to restore its state.

### 3.2 Aggregate Gossip

In an aggregate protocol \[10\], each node is permitted to compute some function of the received state and propagate the result of this function to other nodes. This function may incorporate some information known only to that node (which is never transmitted directly). This behaviour allows the protocol to compute some aggregate function of the information held by each node. For example, if every node holds an integer, the nodes can agree on the maximum value of these integers using an aggregate protocol \[9\]. This is achieved by treating the information propagated as an estimate of the maximum, and by having every node keep a local estimate (set to the value of the node’s integer in the first step). Each node then compares the received estimate with its own estimate and chooses the largest to produce a new estimate of the maximum. As estimates propagate across the network, the estimate of each node will converge on the true value of the maximum, and every node will be aware of it.

An aggregate function (one which converges under these circumstances) must (i) be divisible into pairwise operations (so a new estimate can be computed from an old estimate and local information) and (ii) proceed monotonically towards the true value (according to some distance function) \[18\]. If the aggregate function is non-monotonic (such as if the node alternated between computing a minimum and a maximum) then the estimate will never converge.

### 4. FLASHMOB

Our distributed assembly process, FlashMob, uses an aggregate gossip protocol to enable the set of nodes to derive and agree upon a global component configuration. Each node may host some subset of the components in the configuration, and it may be that only a subset of the nodes in the network are involved in hosting components. For simplicity, all nodes know the user’s functional requirements and non-functional preferences. Only one node need be aware of the structural constraints, though greater performance can be achieved if the information is available to many nodes. No node knows the full set of components available on the network.

Once a configuration is agreed upon, it is instantiated by determining the wiring automatically, connecting each required interface to any suitable provider. If an adaptation is subsequently necessary—due to a change of goal, a component failure or unacceptable NF properties—the gossip protocol resumes to find a new solution.

The gossip protocol for assembly uses a uniform push policy. Each node’s state is an “estimate” of the (global) component configuration, and can use the local component repository to propose a new state. Figure 2 shows an overview of the process in which two nodes exchange a single gossip message, resulting in a change of state. In the following the state representation (and thus the format of gossip messages) is given before describing the rules used to generate a new state.

**Figure 2: Overview of FlashMob**

### 4.1 State Representation

The state of a node is an estimate of the global component configuration. The state is a set of “active” dependencies which represent a decision on which component provisions will satisfy which requirements. Gossip messages contain a single state.

\[
\text{State} \subseteq 2^{\text{Dependency}}
\]

\[
\text{Dependency} \subseteq \{\text{prov, req}\} \times \text{Component} \times \text{InterfaceType}
\]

A Dependency is a tuple of a Component, an InterfaceType and prov or req indicating whether the dependency represents a provision or a requirement. In the case of a provision \((\text{prov, c, i})\), a component \(c\) is present in the configuration and is responsible for providing interface \(i\). A Component \(\subseteq \text{ComponentType} \times \text{NodeIdentifier}\) is in fact a pair comprising the actual implementation type and the node on which the component is hosted.

Given a State, the set of components active in the configuration can be found by collecting the Components mentioned in each Dependency. For example, the state

\[
\{(\text{req, (c, n_0), i}), (\text{prov, (d, n_0), i}),
(\text{req, (d, n_0), j}), (\text{prov, (e, n_1), j})\}
\]

describes a configuration of \((c, d)\) hosted on node \(n_0\) and \(e\) hosted on \(n_1\). Component \(d\) satisfies requirement \(i\) of \(c\) and requires \(j\), satisfied by \(e\).
4.2 State Transformation

Upon receipt of a new state $s$, the node applies the following rules:

1. If the state does not contain one of the functional requirements (interfaces) $i$, then the node adds ($req$, $\ldots$, $i$).

2. If there is a requirement ($req$, $c$, $n_0$, $i$) with no corresponding provision ($prov$, $d$, $n_1$, $i$) and the node $n$ knows a component $c$ which provides interface $i$, then the node can add ($prov$, $c$, $n$, $i$) to the state along with the other provisions and requirements of $c$. This rule leads the configuration towards being complete\(^2\). If there are multiple local providers of $i$, non-functional preferences are used to choose between them.

3. If the state is complete and meets all the functional requirements, the node may evaluate it against the structural constraints using a local constraint checker. If the check fails, the node can

   (a) adopt a randomly-selected previous state (for this purpose, a list of all incomplete states is maintained), which has the effect of backtracking in the hope that a different choice from the previous state will lead to a valid configuration, or

   (b) add a randomly-selected component to the state, which has the effect of performing an exhaustive search through the space of configurations in the hope that a valid configuration will be found. This is sometimes necessary when the structural constraints require a component which has no dependency relationship with the others that have been selected (see [8]).

The new state $s'$ is only accepted if the following conditions hold: (i) $s'$ is not a subset of the node's previous state $s_n$; (ii) $s'$ must be complete and valid if $s_n$ is complete and valid, and (iii) $s' > s_n$ according to the partial order $>$ (we use a lexical ordering). Conditions (i) and (ii) ensure that the algorithm is monotonic: a node cannot accept an incomplete solution when it already knows a more complete solution. Condition (iii) ensures that when there are two possible complete solutions, every node chooses the same one.

In the next section we illustrate the essential features of the protocol before addressing some of the finer points necessary to complete the description.

4.3 Example

Figure 3 shows a system composed of three nodes each of which hosts one component, $A$, $B$ or $C$ (we shall use these names to refer to both the node and the component, using italic type to distinguish the nodes). The interface $ai$ is identified as a functional requirement and is provided by $A$. $A$ also requires $bi$. Thus in step (i) node $A$ identifies a requirement it can satisfy and adds the provision ($prov$, $A$, $ai$) and the requirement ($req$, $A$, $bi$) to the state. $A$ transmits this new state to a randomly-selected node, which in this case is $B$. Likewise, in step (ii), $B$ sees that ($req$, $A$, $bi$) is a requirement which it can satisfy. Component $B$ has the further requirement of $ci$, so ($req$, $B$, $ci$) and ($prov$, $B$, $ci$) are added to the state. This is transmitted back to $A$ ($A$ also transmits its state again). In step (iii), $A$ makes no changes to the state (since it cannot satisfy $ci$) and at some point forwards it to $C$. In step (iv), $C$ satisfies the requirement for $ci$ by adding ($prov$, $C$, $ci$) to the state, and forwarding it arbitrarily to $B$. $B$ has no changes to make and eventually forwards it to $A$. After step (iv) all nodes agree on the state \{$($req, $\ldots$, $ai$)$, ($prov$, $A$, $ai$), ($req$, $A$, $bi$), ($prov$, $B$, $bi$), ($req$, $B$, $ci$), ($prov$, $C$, $ci$)$\} which gives the configuration \{$A$, $B$, $C$\}.

Since the configuration after step (iv) is complete, it can be evaluated against the user’s structural constraints, such as whether the configuration is a pipeline. Any node can perform this evaluation, and in the case of failure revert to some previous state. Unfortunately in this case there are no alternative configurations so the search would fail.

4.4 Enforcing Structural Constraints

Unfortunately, deriving a complete configuration using rules 1 and 2 as in the example does not guarantee that the solution will meet the structural constraints, which can only be checked when the solution is complete. Therefore it is necessary to extend the approach beyond an ordinary aggregate protocol to incorporate backtracking when a complete solution fails the constraint check. This is the purpose of rule 3, which gives the node a choice between (a) backtracking to a previous state and (b) adding an arbitrary component.

In case (a), the node chooses at random an entry from the list of all states it has seen previously, and propagates it. Since this state is likely to be an “ancestor” which led to the derivation of the invalid configuration, it is necessary to prevent the derivation of the invalid configuration by informing the other nodes that it fails the constraint check (in case they have not realised themselves). For this purpose we have adapted the “death certificates” of [4] into a secondary gossip protocol. When a state fails the constraint check, a death certificate is produced and propagated across the network using a uniform push policy. Upon receiving a death certificate, the state mentioned is added to a local list and if the offending state ever recurs, its propagation is suppressed.

The propagation of death certificates forces the algorithm to choose alternative component implementations (by, for example, overriding the $>$ ordering on solutions). Since backtracking is random, eventually either a valid solution will be found, or every node will contain a list of death certificates for every possible configuration.

Notice that death certificates are not strictly necessary if every node has the full set of structural constraints, since every node would build up the list of death certificates locally without any propagation. However, in that case death certificates would work as an optimisation since they can be issued before the network has agreed upon a solution, allowing nodes to dismiss unseen solutions. In other words, using death certificates takes advantage of parallelism in that different solutions can be checked simultaneously by different nodes, as opposed to requiring every node to check every solution.
Case (b) simulates the exhaustive search of the space of components which is performed in the centralised algorithm when no solution in the transitive closure of the dependency relation satisfies the constraints. Again, random choice ensures that eventually a valid solution is found or the full combinatorial space is covered.

4.5 Convergence & Non-Convergence

There are two properties of interest when considering convergence (agreement on a single solution). The first (and most important) is whether the algorithm is guaranteed to converge in all cases. The second is how quickly one can expect convergence to occur, with respect to the size of the problem and the number of nodes involved.

In order to show that convergence is guaranteed, we need to show

(i) that the aggregate protocol proceeds monotonically to a solution and

(ii) that all nodes come to agreement on that solution.

Firstly consider the protocol without backtracking. This restricts our view to rules 1, 2 and 3(b) of Section 4.2. Notice that each of these rules can only add Dependency entries to the state, and so the state can only increase in size. Additionally, if a node receives two updates “out of order” whereby the second update is an ancestor (a subset) of the first, only the first is retained. This means the state cannot decrease in size (unless it is a different solution entirely, in which case we rely on gossip to achieve agreement). Hence, the protocol is monotonic, giving (i).

For (ii), we rely on standard properties of gossip that allow us to assume every node will eventually receive every update (that is not rejected by another node). Then we must ensure that every node chooses the same solution from this “acceptable” set. It is for this purpose that we order solutions using $>$, and have nodes choose the greatest. This leads to agreement, giving (ii).

The effect of (i) and (ii) is a protocol in which the various solutions grow monotonically until completion, at which point each node makes a choice using the arbitrary (but fixed) ordering. However, so far we have omitted backtracking. Clearly backtracking breaks monotonicity, so it is necessary to show that backtracking does not lead to infinite loops. Every time backtracking occurs, a death certificate is produced for the invalid configuration. This is propagated across the network using uniform push (and so all nodes eventually have the death certificate). Since backtracking is performed by making a random choice between the previous states (and all solutions are reachable from the “root” state) we can say that if there is a valid solution, it will eventually be derived (since the probability of backtracking to the necessary state is non-zero). If there is no valid solution, then every node will receive a death certificate for every possible solution, and backtracking will continue indefinitely. A time-out must be used in this case since although a node may have a death certificate for all solutions, there is no way for the node to detect this situation, since the node does not know the whole dependency graph. The unfortunate side effect of a time-out is that the solution may be found just after the time limit, and so the algorithm is no longer complete (in the sense of being able to derive solutions when they exist). Depending on the application domain, it may be a better policy to fall
back to the centralised algorithm after a time-out to ensure completeness.

In summary, without any structural constraints the algorithm is guaranteed to converge on a single solution (even if a requirement cannot be satisfied), and, when structural constraints cause backtracking, the algorithm will find a solution if one exists and if it is found before the time-out.

### 4.6 Time To Convergence

In a uniform push protocol (and in several aggregate protocols [10]), convergence after an update occurs in $O(\log n)$ rounds [4]. Assembly, if backtracking is disregarded for the moment, can be seen as several such updates performed (in the worst case) sequentially. Then for a solution size (number of components$^3$) $s$, convergence can be expected within $O(s \log n)$ rounds. Ordinarily, of course, updates happen in parallel, providing even better performance.

### 4.7 Detecting Convergence

Although the protocol theoretically converges within a certain number of rounds, a mechanism is required for individual nodes to detect that this is truly the case, so that the configuration can be instantiated and the plan can begin execution. The algorithm has converged when

- **C1.** every node has the same state, and
- **C2.** no node can suggest a new state according to rules 1 and 2.

This situation can be detected by each node independently when it

- **L1.** observes a lack of state changes for $O(\log n)$ rounds. If no node can suggest any changes, then the time remaining before convergence is the time it takes for a single update to propagate in a non-aggregate push algorithm, which is $O(\log n)$, and
- **L2.** cannot apply rules 1 and 2 to the current state.

It is trivial to see that if condition L2 holds on all nodes, then C2 holds. To see how condition L1 ensures C1, suppose that C1 does not hold (but C2 does). Then the nodes disagree on the state but (since C2 holds) they cannot suggest any changes. In this case, the nodes will come to agreement within $O(\log n)$ rounds (under the normal expectations of gossip). Thus if C1 does not hold, a node can expect a state change to occur within $O(\log n)$ rounds$^4$, falsifying L1.

When both L1 and L2 hold, a consensus protocol can be used to ensure that C1 and (particularly) C2 hold. Commit protocols such as centralised or decentralised two-phase commit [13] can be used, which have a negligible effect on the number of rounds. Such protocols do however come at some cost in the number of messages sent ($O(n^2)$ in decentralised two-phase commit). Another alternative would be to use the variant of gossip known as rumour mongering whereby, with a certain probability, nodes stop propagating an update after some number of rounds. Eventually, all nodes will

---

$^3$Strictly the solution size is the number of *Dependency* entries but this is usually proportional to the number of components.

$^4$For this reason the number of nodes $n$ is known globally.

---

### 5. EVALUATION

The three properties of interest to us in evaluating this approach are (i) its suitability for application within the three-layer model — for which purpose we use an existing case study from the robotics domain — (ii) its scalability, and (iii) its robustness to node failures and message loss. For the latter two properties we ran FlashMob in simulation (on a single machine) and over the Imperial College network comprising 80 machines.

#### 5.1 Case Study

To see how the approach fits within the three-layer model it is necessary to consider a goal for a particular domain and how self-assembly takes place on that basis. In this case, we choose one particular goal (and suppose the existence of a plan) that requires a number of Koala$^5$ robots to carry objects (balls) from one robot arm to another. The exact locations of the arms are not known, necessitating a search for each arm. This scenario could represent various real-world applications. In a military context, unmanned vehicles could be used to carry supplies across hostile territory, between manned units. Likewise in a disaster situation or a factory setting, the robots might be used to carry objects through dangerous areas. Figure 4 shows the scenario graphically.

---

![Figure 4: Supply scenario](http://www.k-team.com)

---

The software on each of the robots can be assembled from various components which implement fundamental behaviours. Some components are only available on certain nodes in the system. For example, the Koala component provides access to the hardware and must be used only a Koala robot, while the KatanaArm component can only be used on a robot arm. Figure 5 shows all the domain components arranged in a dependency graph. Unfortunately, there is no space to describe them all here.

The functional requirements for our goal are represented by the Surveyor interface (to search for the arms) and the Grabber and Placer interfaces (which respectively provide functionality to grab and place coloured balls). The main point of variation between the possible configurations is in the choice of SearchPattern implementation. Although this fact is not necessarily known at design time, as component
availability changes, we use it to restrict our discussion of non-functional properties to these components. Suppose each search pattern is annotated with the following NF properties:

- CircularSearchPattern: $(\text{efficiency}, 0.8), (\text{success}, 0.5)$.
- ZigZagSearchPattern: $(\text{efficiency}, 0.5), (\text{success}, 0.6)$.
- RandomSearchPattern: $(\text{efficiency}, 0.3), (\text{success}, 0.9)$.

with the intended interpretation that “efficiency” is a measure of how quickly the pattern can find a target, and “success” is a measure of how often the pattern succeeds in discovering the target. For example, RandomSearchPattern has a low efficiency since it causes random motion with no regard to what areas have already been covered, while it has a high success rate since it eventually covers the whole area.

Each property is weighted according to the user’s preferences. In this case, the weights are $w_{\text{efficiency}} = 0.6$ and $w_{\text{success}} = 0.4$, reflecting the user’s preference that the Koalas cross the perilous zone quickly and the positions of the arms (though they are mobile) are expected to be relatively predictable (see [16] for a complete discussion of NF selection).

We can now apply FlashMob to generate a configuration for this application. There are 4 nodes with the following component repositories:

- A Koala $k$, hosting \{Koala, KoalaMotorController, VectorMotionController, ObstacleAvoider, WorldEdgeAvoider, StaticObstacleAvoider, ObstacleMap, CircularSearchPattern, ZigZagSearchPattern, RandomSearchPattern, BallSurveyor, VisualDoorLocator, Webcam, GoToTask, VisualLineFollower\}

- Two arms $a_1, a_2$, each hosting \{KatanaArm, BallGrabber, BallPlacer, Webcam\}

- A location server $s$, hosting \{SkyCamera\}

Each node is provided with the set of functional requirements and relevant structural constraints. The initial state of each node is thus $c_0 = \{(\text{req. Surveyor}), (\text{req. Grabber}), (\text{req. Placer})\}$

The nature of gossip means that there are many possible execution sequences, however we can only consider one here. Suppose in the first step that $k$ applies rule 2 (Section 4.2) to satisfy the Surveyor requirement using the BallSurveyor, and propagates the state $c_1 = c_0 \cup \{(\text{prov. (BallSurveyor, k), Surveyor}), (\text{req. (BallSurveyor, k), Camera}), (\text{req. (BallSurveyor, k), SearchPattern})\}$

Simultaneously, $a_1$ satisfies the Grabber requirement with BallGrabber, and propagates the state $c_2 = c_0 \cup \{(\text{prov. (BallGrabber, a_1), Grabber}), (\text{req. (BallGrabber, a_1), Camera}), (\text{req. (BallGrabber, a_1), RobotArm})\}$

In the next step, $a_1$ receives the state $c_1$ from $k$, and again adds the BallGrabber. $k$ satisfies the new requirements for a SearchPattern and a Camera by adding the Webcam and choosing one of the SearchPattern implementations. The utility of each is calculated according to the user preferences [16]:

$U(\text{CircularSearchPattern}) = 0.8 \times 0.6 + 0.5 \times 0.4 = 0.68$

$U(\text{ZigZagSearchPattern}) = 0.5 \times 0.6 + 0.6 \times 0.4 = 0.54$

$U(\text{RandomSearchPattern}) = 0.3 \times 0.6 + 0.9 \times 0.4 = 0.54$
The CircularSearchPattern is thus selected producing a new state

\[
c_3 = c_1 \cup \{(\text{prov}, \text{Webcam}, k, \text{Camera}),
(\text{prov}, \text{CircularSearchPattern}, k, \text{SearchPattern}),
(\text{req}, \text{CircularSearchPattern}, k, \text{MotionController}),
(\text{req}, \text{CircularSearchPattern}, k, \text{LocationServer})\}
\]

Suppose then that \(a_1\) receives this new state and adds the BallGrabber and the KatanaArm to produce a state

\[
c_4 = c_3 \cup \{(\text{prov}, \text{BallGrabber}, a_1, \text{Grabber}),
(\text{req}, \text{BallGrabber}, a_1, \text{Camera}),
(\text{req}, \text{BallGrabber}, a_1, \text{RobotArm}),
(\text{prov}, \text{KatanaArm}, a_1, \text{RobotArm})\}
\]

When this is received by \(k\) it is accepted as a superset of the state known to \(k\).

In the following step, \(s\) receives the state and satisfies the LocationServer requirement introduced by the CircularSearchPattern with the SkyCamera. \(k\) also satisfies the remaining requirements by adding the VectorMotionController and Koala.

Suppose now that the nodes \(k, s\) and \(a_1\) agree on the state

\[
c_5 = c_4 \cup \{(\text{prov}, \text{SkyCamera}, s, \text{LocationServer}),
(\text{prov}, \text{VectorMotionController}, k, \text{MotionController}),
(\text{req}, \text{VectorMotionController}, k, \text{KoalaMotors}),
(\text{prov}, \text{KoalaMotors}, k, \text{KoalaMotors})\}
\]

which has the remaining requirement of a Placer. This can be satisfied by \(a_1\) or \(a_2\) to produce a complete configuration which can then be checked against the structural constraints. After backtracking to meet those constraints, a complete and valid solution is propagated to all nodes and convergence is detected. The chosen configuration is shown in Figure 6, which additionally shows which nodes host which components. Our implementation assembled the configuration in 28 rounds, taking 2578ms, though the sequence of messages was different from that described above.

As described in [15], these configurations are instantiated and then the "root" components (BallSurveyor, BallGrabber, BallPlacer) are called in order to perform actions in the plan, which has been generated from the goal.

The successful integration of FlashMob with the rest of the three-layer model stems from the isolation of plan generation from the manner of self-assembly. Plans are generated in terms of domain actions\(^6\) implemented by certain interfaces. These interfaces are then the only restriction placed on self-assembly by the goal management layer. Likewise when plan execution begins, the assembled configuration interacts with the plan interpreter through these interfaces. The upshot is that the change management layer is free to perform configuration assembly on any basis.

When adaptation is needed in response to a component failure or environmental change, or if the NF properties of a component diverge from those advertised (as detected through monitoring), a death certificate is issued for the failed configuration (reflecting a change of availability) and gossip resumes from the configuration without the offending component. If further failures happen during the adaptation, gossip will still converge provided that failures do not continue indefinitely. When adaptation is needed to handle changing user requirements (a new goal), gossip proceeds from a new set of interfaces following from the new plan.

### 5.2 Larger Experiments

The chief purpose of using gossip to perform self-assembly is to ensure that the technique scales well to large numbers of nodes and that it is robust to failures. We tested these properties through larger experiments involving either simulated nodes on a single machine or 80 modern desktop machines running Linux 2.6 on an Intel dual-core processor communicating using UDP over gigabit ethernet.

In the first instance we tested the approach on regular, predictable configurations. Using regular configurations allows us to verify the assembled configurations with ease and means that any unexpected results follow from the algorithm itself rather than the peculiarities of the domain used for testing. Initially, the experiments involved no message loss.

![Figure 7: A heterogeneous ring](image)

In the first experiment, the sets of components were structured into rings of increasing sizes. Each component was of a different type, providing and requiring a single interface (in other words, the ring structure was built into the dependencies rather than enforced as a constraint). There were no further dependencies such that there was only a single trivial solution in each run (the number of components \(c\) equals \(s\), the solution size). Components were assigned to nodes in a regular way so that, for example, if

\(^6\) Architectural actions [1, 17] are not included in plans partly for this reason and partly because inclusion would hugely increase the planning state space.
there are 100 nodes and 20 components, then every fifth node is assigned one component. Figure 7 shows a ring of size 4.

Figure 8 shows the number of rounds taken by the algorithm to reach convergence given a ring comprising 20 components, and increasing numbers of nodes. This fits a logarithmic curve, confirming our expectation of convergence within $O(s \log n)$ rounds. Each round took approximately 100ms, and so the actual time taken to converge in each run was between 3 and 15 seconds. The simulated runs give a very similar result.

Figure 8: Ring of size 20, with increasing nodes

Figure 9 shows the behaviour with a fixed number of nodes, and increasing sizes of ring. The relationship is linear as expected ($O(s \log 5)$). The number of rounds is approximately double the size of the ring.

Figure 9: 5 nodes, and increasing sizes of ring (simulation)

Having shown the scalability of the approach on regular configurations, it is desirable to verify that the results extend to general configurations, and in the extreme case, random configurations. The success of the algorithm on random components shows that it is not tuned to specific styles, to configurations of a certain size, nor does it rely on the intuition of the designer, in the way that repair strategies are written to handle a very restricted domain of components.

Figure 10 shows the number of rounds before convergence was detected for increasing numbers of nodes over several runs. 40 randomly-generated components were used as input, with each interface randomly assigned to 4 components on average and no components hosted on more than one node.

Figure 10: 40 random components (simulation)

In this case it is hard to say whether a linear or logarithmic curve fits the results best. Convergence is expected in $O(s \log n)$ rounds, but each run (each point in the graph) has a different $s$ ($1 < s \leq 40$) due to the fact of random generation. This explains a “noisier” curve than with regular structures. In any case the graph shows that the number of rounds grows slowly with the number of nodes.

The final experiment in Figure 11 shows how the protocol copes with message loss. Here, the simulation discards varying proportions of gossip messages from 10% to 85% and the time to convergence is measured. No convergence messages were discarded. The protocol tolerates up to 50% message loss with a small increase in the number of rounds, but this increases sharply such that 85% message loss causes a four- to five-fold increase in the time to convergence.

Figure 11: The effect of increasing message loss, on a ring of size 20 with $n = 5$ (simulation)

6. RELATED WORK

As one of the few works considering distribution, the approach of Georgiadis [7] is a prime candidate for comparison. In that work, each distributed component has a configuration manager which maintains a model of the current (global) configuration and is charged with enforcing architectural constraints provided by the user. Managers receive notifications when the architecture changes and if at any point one of the component’s requirements is not satisfied, then the manager attempts to acquire the change lock and make modifications which result in the satisfaction of the requirement while simultaneously respecting the architectural constraints. These modification scripts are written by the system architect, and in this regard the
work bears great similarity to [6], except in a distributed setting. In contrast, our approach derives configurations in a declarative manner by having the user specify what properties a useful configuration has, rather than specific instructions on the construction of an ideal configuration. This means that our approach affords the system more opportunities for adaptation by going beyond the designer’s expectations.

The success of Georgiadis’ approach hinges on the fact that managers are guaranteed to have a consistent view of the global configuration through the use of reliable, totally-ordered broadcast. This communication mechanism does not scale to large systems. Using gossip improves the situation considerably such that the algorithm can tolerate message loss and node failure without a significant performance loss, and the time taken to reach convergence is logarithmic in the number of nodes, ensuring that the approach will scale to very large networks.

7. CONCLUSIONS

In this paper we have presented FlashMob, a gossip-based technique for dynamic adaptive self-assembly. The approach is capable of deriving and modifying configurations of distributed components so that the functional requirements, non-functional preferences and structural constraints expressed by the user are met initially and continue to be met as the context of execution changes. FlashMob decentralises the process so that no single node is responsible for assembling configurations and maintaining all the knowledge necessary to do so. The use of gossip means that the approach can handle node failure and message loss and can scale to large networks of nodes.

There remain many points of variation for future work, without changing the foundations of the approach. The variant of gossip used (uniform push) is an obvious candidate. A particularly interesting direction would be to consider the effect of different distributions of components on non-functional properties. For example, components which need to interact a great deal are better placed on the same node, if possible, to avoid the performance hit of communicating over the network. But of course, this leads the configuration back towards a centralised solution dependent on the reliability and performance of a single node. This suggests that the trade-off should be made explicit in the assembly process so that, given a choice of where to host a component, the most appropriate option for the current context is chosen.

8. REFERENCES