Quantitative Analysis of Dynamic Reconfiguration Algorithms

Jamie Hillman & Ian Warren
Computing Department, Lancaster University, Lancaster, UK
Email: j.hillman | iw @comp.lancs.ac.uk

Abstract

Safety and mission critical systems often have a requirement for high availability. When the implementation of these systems must be changed for some reason, change must be carried out at run time, which introduces the potential for inconsistency and failures. The field of dynamic reconfiguration has arisen to address these issues. Dynamic reconfiguration algorithms aim to carry out change safely and consistently at run time. Algorithms differ in their approach to managing change and consequently the level of performance they offer. Furthermore, different algorithms impose different constraints on their applicability. We argue that it would be beneficial to be able to compare the behaviour and performance of various algorithms in any given scenario. This paper presents an approach which allows developers to compare dynamic reconfiguration algorithms and to gauge their performance in a given situation, based on simulation and quantitative analysis.

1 Introduction

As software systems are becoming more complex it is increasingly difficult to understand them as a whole. Consequently deploying and operating a software system inevitably reveals issues and problems that were not anticipated at design time. This contributes to the need for change in a software system. Where there is a requirement for the continuous availability of a system (due to safety-critical or mission-critical function for example), change must be carried out *dynamically* at run time.

There is great danger inherent in manipulating a system at run-time. If care is not taken, inconsistencies and failures can occur which would have more harmful consequences in the environment of the system than taking the system entirely offline. Change must therefore *preserve consistency*, that is, change should be carried out without leaving the system in an inconsistent state.

The field of Dynamic Reconfiguration is concerned with carrying out change at run-time, whilst preserving consistency. This field has seen the development of algorithms which move a system into a state where change can be carried out safely. This is usually achieved by temporarily limiting the communication between a component to be changed and those surrounding it, so that change does not take place when interactions are in progress. Algorithms differ in three dimensions; how they move a system into a safe state, what they require from the system in order to achieve this state, and their performance. Performance can be measured in terms of how quickly an algorithm carries out change and how much disturbance is caused to the underlying system. Dynamic reconfiguration algorithms cannot be evaluated absolutely however, they can only effectively be evaluated in the context of a particular application and change scenario.

We argue that it would be of great advantage to be able to observe the performance and behaviour of a range of algorithms given a particular change scenario and application, in order to decide which algorithm to use to manage change. However, existing implementations of dynamic reconfiguration algorithms[6, 7, 12, 9, 1] are closed and embedded in middleware technologies and so such comparison is very difficult.

The remainder of this paper is structured as follows. In section 2, we identify some key differences in existing reconfiguration algorithms. We illustrate some of these differences using two contrasting algorithms: an optimised algorithm and a general purpose algorithm. In section 3, we present a brief overview of the OpenRec framework, which provides support for comparing different reconfiguration algorithms. To demonstrate the value of OpenRec, we describe a component-based network router and present the statistics OpenRec captures as it is used to reconfigure the router, using the two algorithms introduced in section 2. Finally we draw conclusions and identify avenues for further work in section 5.

2 Background

2.1 Reconfiguration Algorithms

As mentioned previously, dynamic reconfiguration algorithms generally achieve safe change by moving the system into a safe state. In this state, change to a particular component can take place because its communication with other components is constrained. For example if component A is connected to components B and C, a change in A can only be carried out safely if B and C are prevented from communicating with it. This is because inconsistencies are usually caused by executing change whilst a component is busy carrying out a request for another component.

Algorithms differ in how they constrain communication between components and for which components they choose to limit communication. In this second respect they can be divided into two categories, dynamic and static. Dynamic algorithms, such as [9, 1] determine the set of components whose communication should be restricted by means of the *actual* interactions taking place in the system at the time change is required. For example a component may not have its communication abilities reduced unless it actually attempts to communicate with the component to be changed. In contrast, static algorithms determine those components to manipulate based on their *potential* to interfere with the component to be changed, regardless of whether they attempt to communicate with it or not.

Algorithms also differ in their applicability constraints, such as whether they require a particular communication paradigm to be in use or require an application to implement certain interfaces to aid reconfiguration. For example Kramer & Magee's algorithm [6] requires application components to provide interfaces that can be used to direct application components into a passive state, where they assert that they will not initiate further communication with their neighbours.

Finally algorithms can be optimised for particular application scenarios. An algorithm can be restricted to use on systems that adhere to a particular architectural style for example. This enables the algorithm to make assumptions about the system being changed and to manage change in a way that would be unsuitable for other types of system.

2.2 Algorithm 1

Mitchell[7] describes an algorithm for replacing a multimedia codec in a chain of codecs processing a stream. The algorithm described here is a generalised

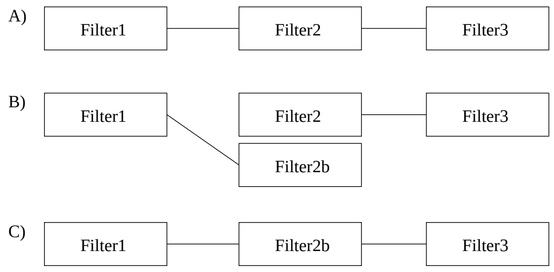

Figure 1: Pipe and filter configuration

version of that algorithm that we have developed to manage reconfiguration of any system conforming to the pipe and filter architectural style[11]. The algorithm essentially functions by running partial chains in parallel and switching between these chains at appropriate times to maintain a consistent flow of data emerging from the last component in the chain. For example to replace *filter2* in the chain of filters shown in Figure 1A (to correct a defect for example), the first step would be to create the replacement filter; *filter2b*. *filter1* would then be disconnected from *filter2* and then connected to *filter2b* so that it can begin processing *Filter1*'s output, as shown in Figure 1B. Then, once *filter3* exhausts the output from *filter2*, it's input is switched to the output of *filter2b*, resulting in the new chain shown in Figure 1C. Using the terminology introduced in the previous section, this algorithm can be classed as static. It has one important applicability constraint which is that the system it reconfigures must conform to the pipe and filter architectural style. Additionally, it requires that filters in the system be stateless, as it carries out reconfiguration actions with no concern for state, and that filters communicate asynchronously. Whilst the case where components are stateless is a rare one, this algorithm serves as an example of an optimisation for a particular domain or application type. For a treatment of the state management problem we refer the interested reader to [4]

2.3 Algorithm 2

Earlier work involved developing a general purpose algorithm[12], which works as follows. Firstly, any component that has a communication path directed to the component to be changed is blocked, causing any requests issued to be queued, a *queueing block*. The component to be changed is blocked using an *abort block*, which causes requests issued to return an error. This prevents the component from making further requests of other components. All that

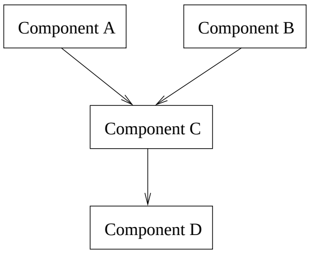

Figure 2: Simple Component Configuration

remains is to wait for interactions that were already in progress to complete. Once this has happened the component can be safely replaced. Once the change has been carried out, the blocks on the connected components are released and the queued requests are issued to the changed component. For example, consider the component configuration shown in Figure 2, where the three components communicate asynchronously. To replace *Component C*, *Component A* and *Component B* would both be blocked with a queueing block and *Component C* would be blocked with an abort block. After waiting for *Component C* to finish processing ongoing requests that may have been received from *Component A* and *Component B*, *Component C* can be replaced. *Component A* and *Component B* would then be unblocked and the system would continue functioning with the new component in place.

In summary, this algorithm can be classified as static and has no significant applicability constraints; it can be applied to systems exhibiting various architectural styles.

3 OpenRec

We have developed OpenRec[1], a framework which facilitates the development, comparison and analysis of dynamic reconfiguration algorithms. Viewed abstractly, Figure 3 shows that the OpenRec architecture comprises 3 layers.

Change Driver The role of this layer is to generate a description of the reconfiguration that is to be carried out on the hosted application. Its implementation is open, the only requirement being that it generates a description of a reconfiguration, expressed in OpenRecML, a language developed for specifying configuration changes. The change driver can be very simple, for example it could generate OpenRecML scripts that cause an application to toggle between two configurations periodically. Alternatively the change driver could be more complex and dynamic, generating a change specification from data received from the system's environment. For example a distributed system might use knowledge of server load to determine whether it should reconfigure itself by creating or destroying server components, to maintain throughput within predefined boundaries.

[1] OpenRec can be obtained at http://www.openrec.org

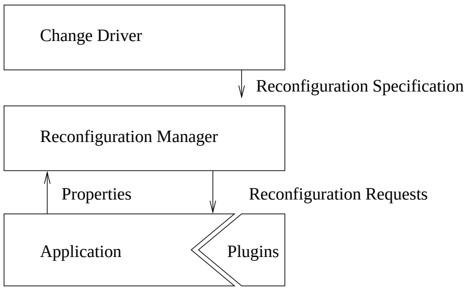

Figure 3: OpenRec Architecture

Reconfiguration Manager This layer controls the reconfiguration process and aims to ensure that change is carried out safely. Given a change specification from the change driver, it will carry out that change on the application, using a specified algorithm. Algorithms are dynamically switchable and extensible. This means that, at run-time, algorithms can be replaced and new algorithms can be introduced.

Application Beyond its support for reconfiguration algorithms OpenRec also serves as a run-time environment for hosting applications. This layer contains the hosted application. The application must be implemented using the OpenRec component model, a reflective component model developed as part of the OpenRec framework.

Being reflective, an application exhibits self-awareness. For example, a particular component can discover which components it is connected to, and the services its neighbours offer. Furthermore, components can be decorated with meta-data which may be useful to reconfiguration algorithms. For example, to use the algorithm developed by Kramer and Magee[6], the Reconfiguration Manager can inspect components to determine whether the necessary change management interfaces are implemented, in order to move them into a passive state (as described in 2.1).

The three architectural levels described above lead to a clean separation of concerns. The algorithm under analysis is separated from both the change driver and the application to be reconfigured. This means that the same change scenario can be executed on the same application with many different algorithms. The behaviour and performance characteristics of each algorithm can be monitored through plugins at the application layer. For example the time during which application components have reduced functionality can be measured, as can the percentage of system components affected by reconfiguration. A key strength of OpenRec is its open architecture. New algorithms, change drivers and plugins can be developed, completely changing the behaviour of OpenRec. Since we are constrained by space here, we direct the reader to [5] for a fuller description of how OpenRec has been implemented and the benefits it offers to dynamic change management.

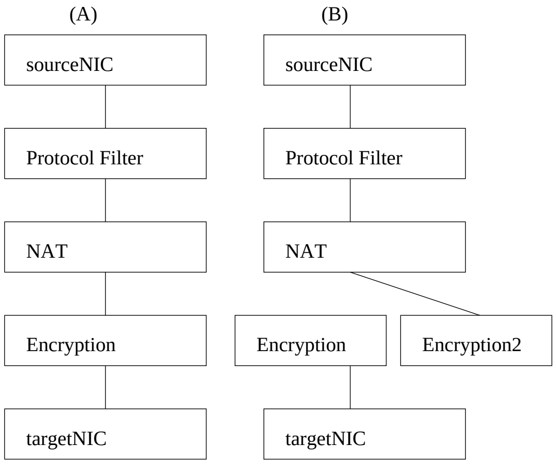

Figure 4: Component-based router

4 Case Study

4.1 Component-based Routing

Current research in the field of network routing is investigating the feasibility of implementing the software embedded in routers using components. The goal is to create extensible and reconfigurable routers that enable new protocols to be deployed quickly and efficiently. LARA++[10] and Netkit[2] are two such research projects, based at Lancaster. Component based routing promises to allow functions that would traditionally be embedded in firmware to be implemented as self-contained interchangeable software components. These components would each perform some distinct function upon packets flowing through the router. Deployment of new routing protocols or routing policies would involve replacing components or adding new components to the component graph. In many situations it may be necessary to replace or remove a component without deactivating the router. In such situations dynamic reconfiguration appears promising for ensuring change is carried out safely.

The system depicted in Figure 4A is a gateway router which would sit between a Corporate LAN and the Internet. It performs three basic functions:

Protocol Filter Ensures protocols that should not be forwarded outside of the corporate LAN are dropped.

NAT Network Address Translator [3], converts internal IP addresses, which are not globally valid, into valid IP addresses.

Encryption Encrypts packets for transport over the Internet to a partner company, which decrypts the packets and forward them onto its LAN.

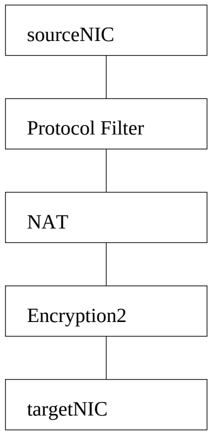

Figure 5: Component-based router - final configuration

Figure 4A also shows two NIC (network interface card) components, which send and receive packets to and from the two networks. Data would flow from the sourceNIC, downstream through the routing components, to the targetNIC.

To highlight the advantages of component based routing, consider a situation in which it is decided the encryption component is too much of a bottleneck and must be replaced with a faster implementation. In a traditional system such a change would involve upgrading the firmware of the router. With component based routing the encryption component can be replaced without such low-level maintenance. Replacing this component whilst the system is running could lead to packet loss or worse and so dynamic reconfiguration techniques are appropriate.

4.2 Reconfiguration Scenario

4.2.1 Algorithm 1

The first action carried out by this algorithm is to create the new component, *Encryption2*. The original component is then be disconnected from it's upstream component, *NAT*. The *NAT* component is then connected to *Encryption2*, resulting in the system state shown in Figure 4B.

The reconfiguration algorithm then waits for *TargetNIC* to starve, that is, it waits until *TargetNIC* stops receiving data from the original encryption component. At this point the output from *Encryption2* is connected to *TargetNIC* in place of that of *Encryption*, resulting in the desired system configuration, as shown in Figure 5.

Since this algorithm is designed for pipe and filter systems, where data flows through a series of filters to an output and continuous data-flow is critical, it aims to preserve data-flow. The replacement component is created and connected to the source early, meaning that it can begin processing data and can generate an output queue. However it's output is not connected to the *TargetNIC* component until the output of the original component is starved. Therefore the replacement component can process its first input item and generate an output at the same time as the original component is generating its final output. Therefore the time taken for the new component to initialise should have a lesser effect on the output of the whole chain because it happens concurrently whilst the old component is still generating output.

4.2.2 Algorithm 2

In our router example, the message flow between *NAT* and *Encryption* is blocked with a queueing block. Messages between *Encryption* and *TargetNIC* are aborted, due to the aborting block in *Encryption*. Once *Encryption*'s ongoing processing completes, it is be replaced with *Encryption2*. The blocks are then removed and queued messages are sent to the replacement component.

As this algorithm is general purpose it does not take any measures to ensure data-flow is preserved through the stream. The original component is blocked and removed before the new component is put in place. We expect this algorithm to cause greater disturbance to the data flow through the chain as the original component must complete its ongoing processing before the new component can initialise and process the first item it receives.

4.3 Results

The router application described in the previous section has been implemented as a simulation using OpenRec's component model. The two algorithms described in the previous section have also been implemented and used to execute the change described in the previous section upon the router configuration. A statistics plugin, which uses the plugin architecture of the OpenRec framework, has been developed to record statistics associated with the reconfiguration.

The key metric in analysing the performance of the two algorithms, when applied to the pipe and filter routing example, is continuity of data-flow. That is, the final component in the chain, *TargetNIC* should ideally be receiving data at the same rate during reconfiguration as it does in the absence of reconfiguration. To determine the normal rate of data flow we simply measure the time a component spends waiting for data from its upstream component (WAIT, for later reference).

Similarly we measure how long a component is unable to receive data as a result of the reconfiguration process. In the case of algorithm 1, the *TargetNIC* component will be unable to receive data because it is in the stage between *Encryption*'s output being exhausted and the subsequent connection to *Encryption2* (see Figure 4B), or because *Encryption2* has not yet generated output. If *Encryption2* has output ready by the time it is connected to *TargetNIC* then the wait time will be the time taken to disconnect *Encryption* and connect *Encryption2* (RECON) in its place. However, in the worst case, *Encryption* could have exhausted instantly and the wait time would be the reconnection time (RECON) plus the time taken for *Encryption2* to process its input and generate its first output (WAIT).

In the case of algorithm 2, *TargetNIC* is unable to receive data because it is blocked (BLOCK). Once *TargetNIC* is unblocked, it will then be subject to the average wait time (WAIT) as *Encryption2* will only just have begun processing.

Figure 6 shows the results of executing the changes described upon the routing simulation, using the two algorithms. The times detailed in these results are not realistic times for a router to exhibit. However, the relative differences between the times is all that is significant here. These differences would remain even if the system was enhanced to function more rapidly.

The results clearly show, as hypothesised earlier, that the parallelism employed by algorithm 1 results in a shorter period of starvation at the end of the chain. The best case time for algorithm 1 is less than a fifth of the normal wait time, due to processing

Application Times	
Time	*Value*
WAIT	49.8ms
Algorithm 1 Times	
Case	*Algorithm 1 Time*
Best Case	8.7ms (RECON)
Worst Case	58.5ms (RECON + WAIT)
Algorithm 2 Times	
Case	*Algorithm 2 Time*
Best Case	50.7ms (BLOCK + WAIT)
Worst Case	100.5ms ((2*WAIT) + BLOCK)

Figure 6: Router Reconfiguration Results

being done in parallel. The worst case time is only 8.7ms extra on the average wait time of 49.8ms, less than one fifth.

Algorithm 2 results in a very large wait time as no processing is done whilst the reconfiguration takes place. In the best case reconfiguration will begin as *Encryption* has finished a transaction and so the time spent waiting for ongoing transactions to complete is avoided, thus the reconfiguration time is the time spent blocking, rebinding and unblocking (BLOCK). After this *TargetNIC* must then wait the average wait time (WAIT) until receiving output from the new component. The worst case for algorithm 2 is when an algorithm has just begun processing and so the total time waiting is the time waiting for ongoing transactions to complete (WAIT), the blocking, rebinding and unblocking time (BLOCK) and the average wait time again (WAIT) as *Encryption2* produces new output. The worst case time is over double the normal wait time, which is a significant overhead.

5 Conclusions and Future Work

The results of the comparative study may have been as easily determined through an analysis of the application scenario described, similar to that conducted in section 4.3. However in many cases it may not be clear which algorithm may perform best from a simple analysis and equally importantly it may not be possible to perform an analysis (for example due to time constraints). This example was chosen for ease of understanding.

This study shows that OpenRec is a valuable tool for developers of reconfigurable systems. Developers are provided with a framework in which new algorithms can be implemented and existing algorithms compared, using a simulation of the target system. Extensible analysis tools reveal the behaviour of algorithms and make it easier to decide which algorithm meets their requirements. In contrast, existing systems provide reconfiguration algorithms to control change but these algorithms are embedded into middleware making comparison very difficult.

The component based router described in this paper, and implemented using OpenRec, is an example of a system which uses a pipe and filter architecture. We have shown that our pipe and filter reconfiguration algorithm causes less disturbance to the router system than a general purpose algorithm, simply because it is optimised for this class of system. We envisage that there are many cases where architecture or domain specific optimisations can be made to reconfiguration algorithms, which would lead to enhanced performance when compared to general purpose algorithms. With our approach, such optimisations can be implemented, observed and quantitatively analysed.

Ideally a knowledge base should exist which indicates which algorithms perform best given a particular change scenario. This would lead to a reduced need for simulation in many cases as the knowledge base could be used to determine the best algorithm to use, based on key system properties. We believe that OpenRec is appropriate for building such a knowledge base and as part of future work we would like to investigate the possibility of creating a change driver which can automatically choose a suitable algorithm for a change, using such a knowledge base. This would be of great benefit for ad-hoc change and for use in self-adapting systems [8].

Finally, we are currently working on advanced tool support for analysing reconfiguration scenarios. For example we are developing visualisation and statistics tools to make reconfiguration more understandable and to give more rich statistics, to further aid the development of dynamic systems.

References

[1] Chen, X., and Simmons, M. Extending RMI to support dynamic reconfiguration of distributed systems. In *22 nd International Conference on Distributed Computing Systems (ICDCS'02)* (2002).

[2] Coulson, G., Blair, G., Gomes, A., Joolia, A., Lee, K., Ueyama, J., and Ye, Y. A reflective middleware-based approach to programmable networking. In *Proc. 2nd Intl.*

Workshop on Reflective and Adaptive Middleware (located with ACM/IFIP/USENIX Middleware 2003), Rio de Janeiro, Brazil, June, 2003.

[3] Egevang, K., and Francis, P. RFC 1631. http://asg.web.cmu.edu/rfc/rfc1631.html.

[4] Feng, N. S-module design for software hot swapping technology. *Technical Report*.

[5] Hillman, J., and Warren, I. An open framework for dynamic reconfiguration. *To Appear at the International Conference on Software Engineering (ICSE) 2004*.

[6] Kramer, J., and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering 16*, 11 (1990), 1293–1306.

[7] Mitchell, S. R. *Dynamic Configuration of Distributed Multimedia Components*. PhD thesis, University of London, 2000.

[8] Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. An architecture-based approach to self-adaptive software. In *IEEE Intelligent Systems, Vol. 14 no. 3, pages 54-62* (1999).

[9] Paula, J., Almeida, A., Wegdam, M., van Sinderen, M., and Nieuwenhuis, L. Transparent dynamic reconfiguration for CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications* (2001).

[10] Schmid, S., Finney, J., Scott, A., and Shepherd, D. Component-based active network architecture. In *In Proceedings of 6th IEEE Symposium on Computers and Communications (ISCC), pages 114-121, Hammamet, Tunisia, July 2001.*

[11] Shaw, M., and Garlan, D. Software architecture: Perspectives on an emerging discipline. *Prentice-Hall, Englewood Cliffs, NJ* (1996).

[12] Warren, I. *A Model for Dynamic Configuration which Preserves Application Integrity*. PhD thesis, Lancaster University, 2000.