2nd International Conference on Autonomous Robots and Agents
December 13-15, 2004 Palmerston North, New Zealand

Design for Dynamic Reconfiguration of Robot Software

Bruce MacDonald, Barry Po-Sheng Hsieh and Ian Warren
Department of Electrical and Computer Engineering, Department of Computer Science,
University of Auckland, New Zealand
B.MacDonald *at* auckland.ac.nz, bshi002 *at* ec.auckland.ac.nz, ian-w *at* cs.auckland.ac.nz

Abstract

Robotic systems must adapt to the changing functionality required of them and to the different real world conditions they encounter, at times requiring dynamic reconfiguration of software components. This paper examines the requirements for dynamic reconfiguration of robotic software in light of techniques used in generic software systems.

Keywords: adaptive robots, dynamic reconfiguration, software engineering, distributed systems, CORBA.

1 Introduction

Our goal is to develop useful robot assistants for humans. Our current robot software design includes a service based, distributed framework in CORBA [1]. Services, such as navigation, motion planning, and localisation, are realised as components that may be physically distributed as CORBA objects. One requirement not yet met is that the software components of a robot system be reconfigured on the fly. While dynamic reconfiguration techniques exist for generic software systems, these do not adequately address the real world requirements for robotics. For example, navigation algorithms may need to be changed as environmental conditions alter during a robot mission, and reconfiguration must meet real-time response requirements for robots.

There is little research on dynamic reconfiguration for robot software. Yu *et al* describe an adaptive middleware system for distributed sensors, aimed at reducing the power consumption by dynamically trading off performance requirements [2]. Gafni considers the architecture design for real-time systems, where the set of active threads must be dynamically reconfigured in response to the sensed conditions [3]. Cobleigh *et al* argue the need for dynamic self–adaption in modules of systems such as robots, in order to provide fault–tolerance [4]. Modules are given the ability to monitor themselves and effect modifications in response to faults. In this paper, we examine the requirements for dynamic reconfiguration of robot software systems, and elaborate on possible design options.

Section 2 gives motivation for dynamic reconfiguration in robotics systems, drawing on our robotics experience. In Section 3 we present key issues for reconfigurable software. We proceed in Section 4 by describing and evaluating design options for a reconfigurable robotic software architecture.

2 Dynamic reconfiguration for robotics systems

Dynamic reconfiguration is important for embedded systems that interact strongly with the real world, since these systems must adapt to changing world conditions and events, and often cannot function correctly if there is downtime. There are two primary motivations for dynamic reconfiguration of robotic software.

Robots must continue to sense, reason and act in real time if they are to avoid obstacles, respond to unexpected physical events, track other moving actors, accurately estimate their locations, and maintain stability in their control systems. This requires that the services accessible to a robot be highly available.

At the same time robots must also be adaptable to the changing needs of their environment. In particular mobile robots may move to new environments that impose different conditions and require different algorithms. For example when a robot moves from a large uncluttered space to a small cluttered room, or from indoors to outdoors, or from a light room to a dark one, the change may be dramatic enough to require different software components for sensing, navigation, and path planning. GPS satellites may become visible outdoors. Rough terrain may require 3D navigation. Infrared cameras may be used when lighting is too dark for a vision system.

To meet these two demands of high availability and adaptability, future robot systems must provide for software reconfiguration. It will not be sufficient to simply initialise a newly deployed software component. In some cases state information such as map data, sensor data history, and position estimates must be transferred to a new software component, such as a new navigation algorithm. The output data stream from a component may need to be maintained when an old component is removed and a new one deployed, for example where the component is providing estimated position data about an obstacle being tracked, perhaps an outdoor vehicle travelling over changing terrain. Algorithm initialisation times must be controlled. Also input data to the component must be managed during reconfiguration, for example the sensor data streams for a navigation algorithm cannot be halted for too long, command inputs must be managed, such as those from the path planner, and data formats must be managed. While we can expect to develop *ad hoc* conversion methods for particular applications such as navigation, it will be more effective to use generic techniques for reconfiguration where we can. Our current robot architecture is a service based distributed infrastructure designed to enhance robot programming, and it will be important to include facilities for dynamic reconfiguration.

A third motivation for dynamic software reconfiguration is that *hardware* systems may be reconfigurable, for example to give redundancy. However unlike many large computer systems with redundant hardware, robot hardware replacements may not be transparent to the software. If the main vision system of a robot fails, a smaller, simpler vision system on a smaller robot may be used instead. Robot hardware changes may necessitate a dynamic change in related software components.

While there has been little attention paid to managing the details of reconfiguring robotics software components on the fly, there has been a history of interest in reconfiguring the physical characteristics of the robot in response to the needs of the task or environment. For example, [5] and [6] present a self–configurable robot design with a distributed software architecture that follows the reconfiguration of hardware. [7] briefly discusses reconfiguration planning for the hardware and connections of modular self–reconfigurable robots.

3 Software dynamic reconfiguration

Component based software provides a sound basis for developing dynamically reconfigurable systems. A component is a unit of composition with well defined provided and required interfaces [8]. A component's provided interfaces represent the services it makes available to other components, while it's required interfaces capture the services the component needs from others in order to function.

A component's interfaces are bound indirectly using connectors. A connector typically implements a particular interaction mechanism, such as (remote) procedure calling or asynchronous messaging. Using any of its bound required interfaces, a component can initiate a request which is handled by the component attached to the other end of the connector. For example, in Figure 1, the localisation component provides two interfaces; one of which is a required interface for path planning and the other a required interface of the emergency stop service. Since path planning and one of emergency stop's required interfaces are bound to localisation's provided interfaces, any requests sent through these interfaces will be routed to the localisation component.

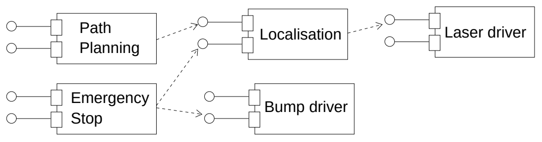

Figure 1: A simple configuration.

Components are thus loosely coupled since they do not hold references to other components. This promotes reuse and configurability since different configurations can be realised using different components and patterns of connection. Component and connector based systems also offer the potential for dynamic reconfiguration where components can be added, removed and replaced at run-time. In addition, connectivity structures between components can be altered dynamically and, assuming a distributed execution environment, components can be migrated between hosts at run-time.

However, dynamic reconfiguration introduces some unique issues and is associated with a generally accepted set of objectives [9, 10, 11, 12, 13, 14]:

Preservation of application consistency. A reconfiguration capability should not corrupt an application. Preserving consistency includes maintaining architectural invariants, synchronising reconfiguration actions with the ongoing execution of the application, and preserving state that is encapsulated in components.

Minimal disruption to the application. During dynamic reconfiguration, only the minimal subset of components should be disrupted from their normal execution. Clearly, replacing an individual component should not involve halting the entire system.

Minimal run-time overhead. Reconfiguration should not impose unnecessary overhead during normal execution nor during reconfiguration. Ideally, a reconfiguration system should incur zero cost when there is no reconfiguration in progress.

Transparency to application developers. Developers should be free to focus on application-oriented issues and should not be burdened with dynamic reconfiguration concerns. Maintaining a separation of these concerns is important to reduce application complexity.

These objectives are ideals and inevitably tradeoffs are necessary. In particular, preserving application consistency will almost certainly incur some run-time overhead and disturbance to application components. To illustrate both the need for managed change and a representative technique for doing so, we show how Kramer and Magee's [11] technique manages replacement of the localisation component in Figure 1. Assume the scenario where path planner has sent a request to the localisation component, which it has started to process but for which it has not yet returned a reply. If the localisation component were destroyed at this point, path planner would likely deadlock waiting for the reply. In addition, since no action is taken to treat any state encapsulated in the localiser, its state will be discarded and thus inaccessible to its replacement.

To address the first problem, Kramer and Magee's approach requires that components implement a change management interface including a passivate operation. In response to a passivate request a component must not generate further outgoing requests. Once the component has finished processing ongoing requests received on its provided interfaces and has received replies from any requests it has made on its required interfaces, the component asserts that it has reached the passive state. Returning to the configuration in Figure 1, localiser would be sent a passivate request in addition to path planner and emergency stop. It is necessary to passivate path planner and emergency stop to prevent them from generating further messages for localiser, since passive components must continue to process incoming requests. Once all three components assert they are passive, the configuration is in a safe state where localiser is quiescent and can safely be withdrawn.

To handle the state encapsulated in the localiser, components may implement unlink and link operations that form part of the change management interface. The reconfiguration management system calls unlink prior to disconnecting the component and link immediately after connecting it's interfaces. In response to an unlink request, a component can divulge interesting state. When the localiser's replacement component is introduced and connected, it can initialise itself with the state externalised by the previous localiser component.

While Kramer and Magee's approach preserves consistency, it can impose a non-minimal level of disturbance on the application. For example, with replacing the localiser in Figure 1, passivating the emergency stop not only prevents it from interacting with the localiser, as intended, but also means that the emergency stop cannot communicate with the bump sensor driver. In this scenario, since neither are being reconfigured, they should ideally be allowed to interact as normal during the replacement of the localiser.

Our final observation of Kramer and Magee's algorithm is that it does not satisfy the objective for transparency. Application components must be implemented to conform with the change management protocol. In particular, when a component is passive it must not use its required interfaces. This conformance is undesirable since it increases component complexity, reduces reuse (existing components cannot be reconfigured using the approach because they do not implement the required protocol), and places part of the burden of reconfiguration correctness on the application programmer. If a single component implements the protocol incorrectly, the effect of reconfiguration will be unpredictable.

More transparent approaches [9, 10, 13, 14] are based on manipulating connectors rather than components. For example, with our earlier work [14], to replace localiser in Figure 1, any incoming requests for it would be buffered, transparently to localiser, by the connectors that link path planner and emergency stop to localiser. Similarly outgoing requests from localiser would be disallowed by the connector that links it to laser driver. Localiser is deemed to have reached the safe state when it is not engaged in processing for path planner or emergency stop and is not waiting for a reply to any request it sent to laser driver.

To handle run-time dependencies between components, [9] and [10] employ run-time analysis of interactions to selectively determine which interactions should be allowed to proceed, in order to reach a safe state for change, and which to block until reconfiguration is completed. For example, assume that to process a request for the path planner, the localiser must send a request to the laser driver; the path planner's request is thus a *dependent* interaction. Once the localiser has started to process the path planner's request, blocking it's outgoing connection with the laser driver must be delayed until after the localiser has sent the consequent request to the laser driver and received the reply. Run-time analysis manages interaction dependencies between components transparently but incurs additional overhead since interactions must be monitored and processed.

4 Reconfigurable architectures

In general, a reconfigurable software architecture for robotics applications should strive to meet *all* the objectives for dynamic change management (Section 3). Preservation of application consistency is universally important for any class of application. Robot malfunctions have caused the death of humans; in such environments the importance of this objective is compounded.

The real-time nature of robotics applications also means that minimising run-time overhead is a significant goal. While it is a common misconception that real-time software must run as fast as possible, when in fact the software need run only sufficiently fast so as to meet the application's timing constraints, the reconfiguration management software should consume as few resources[1] as possible, and should

[1]Including processor cycles, memory and network bandwidth.

itself meet real-time requirements for changes to the configuration. Consequently, more of the system's resources are available to the hosted application, which in turn increases the capacity for meeting the timing requirements of tasks.

Minimising disruption to the application is also more important for real-time systems than with conventional applications. Where application components are unnecessarily prevented from processing normally during reconfiguration, there is a higher risk that these components will fail to meet their real-time constraints.

Finally, realising the objective of reconfiguration transparency is important to leverage our service-oriented framework. The framework is used by robotics application developers, many of whom have limited software engineering expertise. For this reason, they should not be burdened with the additional complexity of dynamic reconfiguration. In addition, as pointed out in Section 3 approaches that rely heavily on application contribution to the change management process increase the risk of reconfiguration failing, with unpredictable consequences. Furthermore, we have a library of service objects that should be reused, with little or no modification, as reconfigurable services.

To help meet all objectives, we have introduced two simplifying assumptions as design constraints:

Services communicate using only independent interactions. By independent interactions, we mean that a service does not make requests of other services in order to process an incoming request. In other words, a service can satisfy a request without needing to delegate work to others. This essentially means that there are no complex run-time dependencies between services and consequently the overhead of run-time interaction analysis can be avoided completely. As reported earlier, run-time monitoring introduces an overhead during normal periods of operation and additional overhead is incurred from analysing interactions when synchronising change with an application.

Reconfiguration management need only support component replacement. With the CORBA model connectivity between client applications and server objects is inherently dynamic and requires no special support from reconfiguration management. Client applications can routinely change the server objects they connect to by acquiring new object references from a naming or trading service. Similarly, new server objects can make themselves available dynamically by registering with these services. Dynamic object replacement, however, requires specific support.

In our experience of writing services using our framework, they tend to be coarse grained and so the restriction of reconfigurable services communicating using only independent invocations is not unreasonable. Similarly, a capability for replacing services is sufficient for our work. Using the service-oriented framework, small robots often interact with services hosted on a limited number of high specification machines. A service such as navigation must always be present, and to support adaptive applications we simply require that its implementation can be changed at run-time. Service removal and migration are superfluous to our needs.

4.1 Design options

The CORBA model fits neatly with the principles of components and connectors introduced in Section 3. Fundamental to CORBA is the interoperable object reference (IOR)[15] which provides for location transparency. An IOR encapsulates the information necessary to locate and communicate with a remote object. It is essentially a proxy that shields client applications from the details of distribution. An IOR can be viewed as a connector, with procedure call semantics, that mediates communication between components.

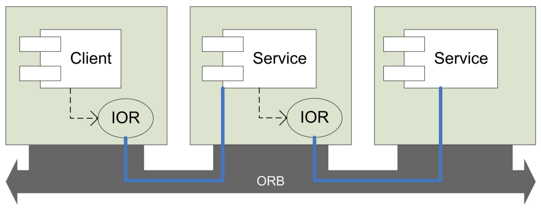

Figure 2: A component configuration in CORBA

Figure 2 shows the structure of a CORBA application involving a client program that is connected to a service component which in turn is connected to another service component[2]. To replace the first service component, subsequently referred to as target, there are two actions required to synchronise the target: *buffer incoming requests to target* and *abort outgoing requests from target*. The problem is thus to perform these actions in a manner which satisfies the four objectives of dynamic change management.

Option 1: A change management protocol. One option is to have components implement an operation with semantics similar to passivate in Kramer and Magee's approach. This is the approach adopted by Almeida *et al* [9] in their implementation of a reconfiguration management service for CORBA.

Option 2: Specialised IOR proxies. An alternative is to employ specialised IORs that can be requested by the reconfiguration manager service to buffer or abort requests. Specifically, client's IOR for target could be requested to buffer outgoing requests until target has

[2]The figure should not be interpreted as the first service sends a request to the second service in response to a client invocation. This would be a dependent interaction. The first service may actively initiate communication with the second service, but not in response to a client request.

been replaced. Similarly, target's IOR could be instructed to abort outgoing calls. The IOR must monitor outgoing calls and their associated replies; when all replies have been received the target is in a safe state.

Option 3: CORBA interceptors. The third option involves the use of CORBA's interceptor facility. An interceptor is a simple reflective mechanism that allows an ORB's functionality to be extended. Interceptors can be used to intercept incoming and outgoing requests. To replace the target service, an interceptor could be introduced through which all incoming requests for the target must pass. In the case where the target service is to be replaced, the interceptor would buffer all requests for the target. To treat outgoing requests from the target service, an interceptor could again be used. In this case the interceptor would abort any newly initiated invocations from the target service by throwing an exception back to the target. Figure 3 (left) illustrates the structure of a server using interceptors. The interceptor must monitor outgoing calls and their associated replies; when all replies have been received the target is in a safe state for replacement.

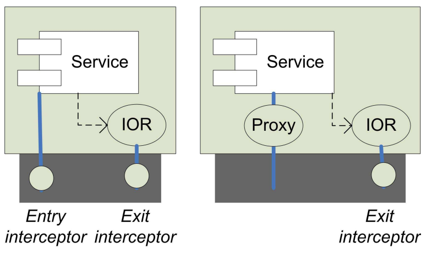

Figure 3: Interceptors and proxies.

Option 4: Interceptors and proxies. A variation on option three is to use proxy objects, as shown in Figure 3 (right), to handle incoming requests to reconfigurable services. With this design, client IORs are connected to proxy objects rather than to service components directly. Similarly to the interceptor design, the proxy objects buffer incoming calls during reconfiguration. An interceptor is still required to trap and monitor outgoing requests because standard IORs are employed.

4.2 Discussion

The first option is unattractive because it does not satisfy the last objective, transparency to developers.

The second design is used by Chen [10] in his dynamic reconfiguration service for Java RMI. For CORBA, however, this approach suffers from two significant drawbacks. First, this design involves working outside of the CORBA specification since IOR implementations are opaque and unspecified by CORBA. Consequently, IORs specialised for a particular ORB are specific to that ORB. This is undesirable since it precludes interoperability across ORB vendors and would in effect lock the reconfiguration management service into a particular product. Furthermore, clients of services using specialised IORs would need server-side capabilities for the IORs to respond to buffer and abort requests. For small robots with few onboard resources this might be impractical.

The second problem with option two is that it requires knowledge of which components are using a particular service since the IORs held by target's clients must all be sent a buffer request. To track which IORs are held by which components would necessitate run-time overhead. For example, a specialised IOR could register itself with the reconfiguration management service and identify the reconfigurable service it acts as a proxy for. A third drawback is that reconfiguration management consumes additional bandwidth since buffer requests need to be broadcast to all IORs for a target service.

Like options one and two, the third design preserves application consistency, subject to the constraint of invocations being independent. However, unlike the first design this approach is transparent to application developers. Furthermore, since incoming requests are buffered on the server side, design three is in contrast to the second design since it avoids broadcasting buffer messages to clients and therefore preserves bandwidth. Application disruption, similarly to the first two designs is restricted to the target component. In all cases, clients will experience a delay in receiving their reply while the target is being replaced, but they are not prevented from communicating with other services. It is only clients' connections with the target component that are blocked.

The interceptor-based solution of design 3 does however incur more than minimal overhead. Interceptors trap *all* incoming and outgoing requests on the machine on which interceptors are active. Since a target service may be collocated with other reconfigurable and non-reconfigurable components, requests to and from these other components will be intercepted too. Proxies, in design four, incur the indirection of interceptors but only for reconfigurable services. Incoming requests to collocated non-reconfigurable services are no longer intercepted with this design, so the introduction of reconfiguration management has zero cost when communicating with conventional services.

Whether the reduction in run-time overhead of using finer-grained proxies over the intercept-all approach of design three is significant remains to be seen. Based on the characteristics of the four design options, we have chosen to implement designs three and four and expect to report on their performance in the near future. We are now implementing these designs using TAO [16]. Currently multiple navigation components are initialised at startup, and one is selected while the others remain idle. On request the manager blocks sensor communication and changes the navigation component.

With all design options, developers of reconfigurable services need to implement a small and simple change management interface comprising two operations: initialise and finalise. The former gives a replacement service the opportunity to initialise its state and takes a parameter which can be the data externalised from another service. The finalise operation allows a component to externalise its state and generally relinquish any resources used. These operations have no impact on the correctness of the synchronisation mechanism.

5 Conclusions

In this paper, we have established dynamic software reconfiguration as an enabler for adaptive robot behaviour and for ensuring that any services required by robots are highly available. We have reported that a dynamic reconfiguration management system should be consistency preserving, transparent to application developers, minimise application disturbance, and incur minimal run-time overhead. For robotics applications, their real-time nature means that the latter two issues are particularly significant.

We have outlined four design options for integrating support for dynamic reconfiguration with our service-oriented framework. While our favoured design options incur a very low level of application disturbance and run-time overhead, we are interested in exploring optimisations. For example, where a service does not encapsulate state, its replacement can be introduced and run in parallel. Because there are no data dependencies between the two services, the original service can complete processing of any ongoing requests, before being removed, while the replacement simultaneously handles newly initiated requests. This optimisation is of particular interest to real-time systems since it means that a service abstraction is always available to process requests, even when it is being reconfigured.

Beyond such optimisations, we intend to investigate how to guarantee that reconfiguration will not break an application's real-time requirements. Currently, our approach is best effort and suitable for soft real-time systems. For applications with hard timing constraints, reconfiguration must be carried out so that all timing constraints are satisfied. This requires knowledge of an application's timing requirements, its run-time timing behaviour and the time required to carry out a particular reconfiguration.

References

[1] E. Woo, B. A. MacDonald, and Felix Trepanier. Distributed Mobile Robot Application Infrastructure. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1475–80, Las Vegas, October 2003.

[2] Xingbo Yu, Koushik Niyogi, Sharad Mehrotra, and Nalini Venkatasubramanian. Adaptive middleware for distributed sensor environments. `dsonline.computer.org/WIPs/May%20WiPs/yu.htm`, May 2003. 2pp. IEEE distributed systems online.

[3] Vered Gafni. Robots: a real–time systems architectural style. In *Proc 7th European software engineering conference*, pages 57–74, Toulouse, 1999.

[4] Jamieson M. Cobleigh, Leon J. Osterweil, Alexander Wise, and Barbara Staudt Lerner. Containment units: A hierarchically composable architecture for adaptive systems. *ACM SIGSOFT Software Engineering Notes*, 27(6):159–65, November 2002.

[5] E. Yoshida, S. Murata, A. Kamimura, K. Tomita, H. Kurokawa, and S.; Kokaji. Self–reconfi gurable modular robots — hardware and software development in aist. In *Proceedings, IEEE International Conference on Robotics, Intelligent Systems and Signal Processing*, volume 1, pages 339–346, October 8–13 2003.

[6] T. Fukuda and S. Nakagawa. Dynamically reconfi gurable robotic system. In *Proceedings, IEEE International Conference on Robotics and Automation*, volume 3, pages 1581–1586, 24-29 April 1988.

[7] Y. Zhang, K.D. Roufas, and M. Yim. Software architecture for modular self-reconfi gurable robots. In *Proceedings, IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 4, pages 2355–2360, 29 October – 3 November 2001.

[8] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. *Addison Wesley*, 1998.

[9] Joao Paula A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert Nieuwenhuis. Transparent Dynamic Reconfi guration for CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, 2001.

[10] Xuejun Chen and Martin Simmons. Extending RMI to Support Dynamic Reconfi guration of Distributed Systems. In *22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[11] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Software Engineering*, 16(11):1293–1306, 1990.

[12] K. M. Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College, 1999.

[13] M. Wermelinger. A Hierarchical Architecture Model for Dynamic Reconfi guration. In *2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE97)*, 1997.

[14] Ian Warren. *A Model for Dynamic Configuration which Preserves Application Integrity*. PhD thesis, Lancaster University, 2001.

[15] Michi Henning. Binding, Migration, and Scalability in CORBA. *Comm. of the ACM*, 41(10):62–71, 1998.

[16] Douglas Schmidt. `www.cs.wustl.edu/~schmidt/TAO.html`.