

Support for Dynamic Distribution in Component Systems

Michael Richmond

mar@mri.mq.edu.au

Microsoft Research Institute

School of Mathematics, Physics, Computing and Electronics

Macquarie University

Sydney, Australia

Abstract

Component technology has been advocated as a means of simplifying the design and implementation of distributed applications. This paper presents a classification of distribution support at the level of middleware component systems.

This classification identifies a form of distribution support, component migration, which is not supported by the component infrastructure currently available. After discussing the benefits to be gained from component migration, we adapt techniques from process migration research to the problem of component migration. Finally we identify a number of open questions that we expect our current work will address.

Introduction

Distributed applications have historically been difficult to build. Typically such applications are large and complex, leading to code reuse and maintenance problems. Component technology promises to simplify distributed application design by encapsulating dependencies within components. Component middleware implementations effectively separate the design of an application from the details of distribution by providing a communication mechanism to the component programmer. We present a classification of distribution support for component systems in terms of increasing application flexibility. After presenting this classification, the provision of component relocation support in existing infrastructure is considered. Emphasis is given to the adaptation of process migration techniques from the operating systems community to component systems.

Distributed Applications

Distributed applications are gaining popularity as they have the potential to provide greater reliability and availability than traditional centralised applications. When a module in a centralised application becomes corrupted, either through software or hardware failures, access to the entire application is lost. Alternatively, if the application were distributed across a number of machines ideally only access to the functions performed by the damaged module would be affected. Fragile or heavily used modules may be replicated in distributed applications to ensure continued availability in the face of failures ie. fault-tolerance.

Such replication may also be used to increase the transaction capacity and throughput of an application by having users introduce additional module replicas to handle growth in transaction rates. This is particularly applicable to systems where the transaction loads are variable. The potential for increasing the available computing resources to an application is a key reason why distributed applications are gaining popularity.

Some applications such as airline ticketing and banking systems are inherently distributed. It is conjectured that these systems are therefore easier to design, build and maintain if the

distribution inherent in the problem is reflected in the resultant application. It is desirable however, to shield the application programmer from the complexities of implementing the support for distribution. Such support for distribution needs to be provided by the underlying component infrastructure.

Recent popularisation of the Internet has ensured the widespread availability of long distance networking. Continued expansion of the Internet and increasing interest in networking solutions in general suggest that the number of computers connected to networks will continue to increase. Since network connectivity is the basic requirement for distribution, current indications suggest the use of distributed applications will increase in the future. Nevertheless, although the hardware infrastructure required for building distributed applications is reasonably mature, it is still unclear how we can best support these distributed applications.

Distribution Support Classification

Currently available component infrastructure such as CORBA [OMG98] and DCOM [Roy97] support distributed applications by providing a transparent communication mechanism between components across hosts. The locality of the components in these systems is specified prior to the execution of the application. Existing distribution support explicitly prevents the re-distribution of components after binding the component.

We conjecture that distributed applications often have a relatively long execution time. It would therefore be desirable for such applications to be adaptive to changing application loads and usage patterns. Such distributed applications require the ability to re-distribute components at any time during the execution of an application.

Without the provision of dynamic support for these changes it is necessary to restart the application when structural changes to the application are made. We have developed the following taxonomy to consider the possible alternatives in specifying the locality of application components:

- **Static configuration**
With statically configured distribution, the locality of the components comprising the application is specified and fixed prior to the execution of the application. This would usually be achieved by creating a configuration file that specifies the locality of each component in the application. Alternatively, during the initial boot, the application could require the user to specify the locality of each individual component.
- **Dynamic configuration (boot-time configuration)**
Dynamically configured distributed applications rely on the component infrastructure to determine the appropriate fixed location for each component in the application. Once the locality of a component is established it does not change for the remainder of the application's execution. Existing systems achieve this by using "trader services". These services allow the application to make a request of the service for a particular type of component. The trader service then performs some form of look-up for a component of the desired type and returns a reference to the appropriate component. This look-up may result in the trader instantiating a component of the required type where necessary.
- **Dynamic relocation (component relocation)**
The initial configuration of a dynamically relocatable application may be specified by either of the preceding methods (or a combination of both). Unlike the previous methods, this form of locality specification is not fixed for the duration of the application execution. This means that the location of a component may be altered at run-time thereby allowing the application to take advantage of changing system loads and adapt to varying application use.

Dynamic Distribution Support

As stated earlier, existing component systems, such as DCOM and CORBA, are sufficient to support statically distributed component-based applications. That is, these systems support applications whose distribution structure is configured prior to execution, with the application's structure remaining static over the entire execution time of the application. In our classification this form of distribution support is identified as static configuration.

The addition of a trader service to the existing component systems introduces support for dynamic configuration of distributed applications. With this type of distribution configuration the application makes a request of a well-known service for a component that meets some specified criteria. On receiving the request, the trader locates an instance of an appropriate component returning a component reference to the application — thereby binding the application to the component. Depending on the criteria specified and the state of the component system a trader may act to create an instance of the required component in order to service the request. This form of distribution support is referred to as dynamic configuration in our classification and is provided in existing component systems by the Object Trading Service from the CORBAServices for CORBA [OMG97].

Clearly each of these options requires an increased level of support from the underlying component framework. Statically configured applications require support to instantiate components on remote hosts according to a specified locality configuration. For dynamically configured applications the framework must support some way of algorithmically determining the best location for each component. This determination may be based on the load of individual hosts in the system, known behaviour of the application (from previous executions/modelling) and the existence of other instances of the required component already in the system.

The third category in our classification, dynamic relocation, requires the component infrastructure to support some form of component migration. At present, it is not clear what form such support will take in the component system. We expect however, that component relocation will be amicable to techniques applied to process migration technology in previous research [RicH97, Ric96].

Component Migration

We define component migration as the movement of an instance of a component from one host in the system to another. Component migration may be used to improve the performance and flexibility of an application in a number of areas. For example:

- **Application load balancing**
In a distributed application the transaction load on given components and hosts often vary significantly. In such cases a component may be migrated from a host with a relatively high load to one under a lower load to improve the performance of the application. The movement of a highly loaded component to a host with a lower load additionally may improve the system performance as a whole.
- **Application restructuring**
The usage pattern and processing load experienced by a distributed application is likely to change over the lifetime of an execution instance. For many distributed applications it may be preferable to restructure the application to reflect these changes without halting the execution of the application. Possible changes to the structure of a distributed application include those resulting from the consolidation of processing nodes in the system; the reintegration of stand-alone distributed components; and the introduction of component replicas to the system. These may all benefit from component migration.

- **Component survivability**
At times the impending shutdown or removal of a host from a distributed application's pool may be known in advance (such as for periodic maintenance, or allocating a host to other tasks). In these cases, a component may be migrated to another host to ensure that the component survives the shutdown or removal of its current host from the application.
- **Reducing network costs**
In some situations it may be preferable to relocate a relatively small component to the site of a resource which cannot be moved (such as a particular device) or a resource whose movement would cause more traffic than migrating the component (such as a large database). When the component no longer requires the resource it may be migrated back to the original host. This is similar in effect to data collection agents with the exception that the control of migration is maintained outside the mobile entity.

The decisions as to when it would be useful to migrate a particular component and which host should be the destination of the migration are system policy decisions and are outside the scope of this paper. Before these policy issues can be addressed we need to establish an understanding how component migration is used in practice. Through the study and use of component migration in real applications we plan to determine which of these policy decisions are best automated and which are better left under user control.

It is clear that though process migration techniques are likely to be applicable to component migration, it is unlikely that any single migration strategy will be universally applicable. For the purpose of migration it is expedient to break a component into three distinct parts:

- The interfaces, type information and structure of the component (ie. the meta-data of the component),
- The code blocks which implement methods of the component,
- The data elements held by the component.

The order in which these parts are transferred from source to destination host by the migration mechanism results in different availability and access properties being exhibited by the component undergoing migration. By studying the behaviour of various migration algorithms produced by accepting different trade-offs in algorithm design we can determine the appropriate algorithm for a given component.

There are two factors that will determine how appropriate each migration strategy is for a given component. First, whether the component contains state which affects the behaviour of future transactions, ie. the component is stateful. Second, whether the structure and method code of the component is available at the destination machine.

During process migration the entire state of the process must be transferred. With some types of component however there is no meaningful state held by the component between method invocations, as is the case with service components. In this case the preferred strategy would be to dispose of the component instance then create a new instance on the destination host. If the component is stateful, that is, if meaningful state is held between method invocations, it is necessary to relocate the state to a new instance of the component to the destination host.

With some component applications it is likely that the code and structure of each component type would be available on all hosts in the system. In this case it makes sense to use the copy of the component code at the destination to build the component at the destination rather than migrating the code. On the other hand, if the component code is not available at the destination we clearly need to migrate the component's code together with any relevant state. This contrasts with the approach used with process migration where the code of the process is regarded as an integral part of the process' state.

Component Migration vs Process Migration

One of the main difficulties that have hindered the wide acceptance of process migration in computing systems is the requirement for location transparency at the process level. In practice it is not possible to include location transparency to an operating system at such a fundamental level without re-designing much of the system. Component infrastructure has been designed to include such location transparency from very early in the design process. Changes to the component system are likely to be less traumatic, for the end user, than changes to the operating system being used.

Process migration has been criticised as being a solution looking for a problem. The lower cost/benefit ratio involved with introducing migration to component infrastructure may mean that component relocation is the problem process migration techniques have been looking for. The OMG (Object Management Group) provide support for component migration in the form of message forwarding in the current standard for CORBA services [OMG97]. CORBA's GIOP (General Inter-ORB Protocol) specifies support and behaviour semantics to provide message forwarding when a component has changed location after bind-time. However, no mechanism is defined to perform the actual relocation of a component.

Component Migration Algorithms

Using our previously developed classification of process migration as a starting point we can identify four algorithms that may be useful for component migration [Rich97];

- Eager-Copy
- Lazy-Copy
- Pre-Copy
- Post-Copy

Eager-Copy

For Eager-Copy, the tasks which must be undertaken to migrate a component would be performed in the following order:

- The decision to migrate the component is made.
- Access to the component is restricted to the migration mechanism.
- The entire state of the component is transferred. This is comprised of the interfaces, the type information and structure associated with the component (i.e. the meta-data), the code blocks implementing the methods of the component, and the data elements held by the component.
- The state of the component is reconstructed on the destination host.
- The imposed restriction on access to the component is removed.
- The component instance on the source host is destroyed.

Eager-Copy is conceptually the simplest migration algorithm, lending itself to a relatively straightforward implementation. Its main weakness is the length of time in which the component is unavailable to clients of the component. For small components however, this delay may be only that incurred by the transfer of a few network packets so may be acceptable.

A variant of Eager-Copy known as flushing may be implemented using a trader service. In this form of the algorithm the state of the component is flushed to a backing store by the trader. A new instance of the component is then created on the destination host and initialised with the state from the backing store. For applications where components are small and share a common local file system this may be appropriate. Flushing is inappropriate however, in situations where components are relatively large and the application is distributed widely, thereby having no common local store.

Lazy-Copy

Eager-Copy transfers all the information for a component before making the component globally available at the destination host. To contrast Lazy-Copy, transfers only the minimum information necessary to allow access to the component via the destination machine. Specifically, the information required would usually consist of the component interfaces, the type information, and structure of the component (i.e. the meta-data). All remaining state is transferred from the source host only when actually referenced by a client request. The order of the tasks is as follows:

- The decision to migrate the component is made.
- Access to the component is restricted to the migration mechanism.
- The minimal subset of the component's state to the destination host.
- The state of the component is reconstructed on the destination host.
- The imposed restriction on access to the component is removed.
- Residual state on the source host is transferred on demand, as required to service client requests.

After global access to the component is restored on the destination host any client request that references state still residing on the source host is trapped by the component system. The required data is then requested from the source host. The client's request is allowed to continue to completion once the required data has been made available at the destination host.

Since the state of the component is only transferred as required there is no general method for determining when any particular portion of the state will be transferred. This has the potential result of the component's state being spread across several hosts in the system. Additionally, the continued availability of the component is now dependent on the liveness and connection properties of the hosts involved. This is known as the residual dependency problem [DouO91].

We expect Lazy-Copy to be advantageous in a component system when the migration is being performed in order to temporarily access a remote resource prior to the component returning to the source host.

Pre-Copy

In contrast to the previous two algorithms, pre-copy does not restrict global access to the component until a complete replica of the component (from some point during its' instantiation) is available at the destination host.

Pre-copy is performed as follows:

- The decision to migrate the component is made.
- The entire state of the component is transferred. This is comprised of the interfaces, type information and structure of the component (the meta-data), the code blocks implementing the methods of the component and the data elements held by the component.
- The state of the component is reconstructed on the destination host.
- Access to the component is restricted to the migration mechanism.
- Any changes to the component's state made after the initial transfer is sent to and integrated into the component's state at the destination host.
- The imposed restriction on access to the component is removed.

There is no way of determining in general whether particular component state will be changed on the source host after its initial transfer. This means that the entire component state must be

retained on the source host until the migration is complete. In the degenerate case, where a component is comprised of only data elements, with every element being modified after its initial transfer, Pre-Copy will exhibit behaviour close to that of a double Eager-Copy.

For large stateless components however, Pre-Copy is likely to be the algorithm of choice as it minimises the time in which access to the component is restricted to the migration mechanism.

Post-Copy

Post-Copy displays behavioural similarities to previous two algorithms. As with Lazy-Copy, access to the component is restored as soon in the minimum possible time after the migration decision is made. In common with Pre-Copy, the state transfer is performed in parallel with continued global access to the component but the component is available from the destination rather than the source host.

Briefly, Post-Copy is carried out as follows:

- The decision to migrate the component is made.
- Access to the component is restricted to the migration mechanism.
- The minimal subset of the component's state to the destination host.
- The state of the component is reconstructed on the destination host.
- The imposed restriction on access to the component is removed.
- The source host transfers the residual component state to the destination. In parallel, the source host services state requests on demand from the destination host that result from client requests of the component.

After the component's entire state (as held by the source host) has been transferred, the component may be discarded at the source host. During the state transfer priority must be given to servicing the demand requests to minimise the delay incurred by clients of the component. These demand requests are performed by ensuring that the component system traps client requests which reference component state that is held at the source host in the same manner as with Lazy-Copy.

Post-Copy is most applicable to the migration of large stateful components, as minimises the period of restricted access to the component and guarantees the completion of the migration after some finite time. This contrasts with the alternatives that either incur long access delays when the component is stateful, or result in residual dependencies on the source host.

Conclusion

Component technology promises to simplify the design and implementation of distributed applications by encapsulating dependencies within components and furnishing the programmer with distribution support. In current component infrastructure this support takes the form of making component communication and distribution mechanisms transparently available on multiple hosts.

We have presented a classification of distribution support that divides distribution support into three categories each providing increasing distribution flexibility at the expense of increasing complexity. We argue that a general-purpose component infrastructure should provide mechanisms to support each of these forms of distribution.

This classification introduces the idea of component migration as a means for providing support for the restructuring of an application without halting its execution. Many similarities exist between component migration and the previously researched field of process migration.

We have adapted the techniques used in process migration to components and presented a brief overview of the issues raised.

Future work includes the extension of existing component infrastructure to support component migration followed by a study of the potential impact of component migration on application design, implementation and use. Some open questions include:

- Is partial access to a component undergoing migration desirable?
- Should client requests to a component be queued until the component has been fully migrated?
- Should migration be limited to a specified component or is extended component migration desirable? (Shallow or deep migration?)
- What user level support is required for component infrastructure that supports component migration?

References

- [DouO91] Douglis, F. and Osterhout, J. *Transparent Process Migration Design Alternatives and the Sprite Implementation*. *Software - Practice and Experience*, 21(8), pp. 757-786, August, 1991.
- [HarC95] Harrison, C., Chess, D. & Kershenbaum A. *Mobile Agents: Are they a good idea?*. Technical Report, IBM T.J. Watson, 1995.
- [OMG97] Object Management Group. *CORBA services: Common Object Services Specification*. Massachusetts, November 1997.
- [OMG98] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.2. Massachusetts, February 1998.
- [Ric96] Richmond, M. *Post-Copy Migration: A new process migration algorithm*. Honours Thesis, Sydney University Computing Department, Sydney, 1996.
- [RicH97] Richmond, M. and Hitchens, M. *A New Process Migration Algorithm*. *Operating Systems Review*, vol. 31, no. 1, January 1997, pp. 31-42.
- [Roy97] Roy, M. & Ewald, A. *Inside DCOM*. *DBMS*, April 1997, pp. 27-34.