# A Framework for Quiescence Management in Support of Reconfigurable Multithreaded Component-based Systems

*Petros Pissias and Geoff Coulson*

Computing Department, Lancaster University, UK

contact: geoff@comp.lancs.ac.uk

**Abstract**

*In component-based software systems the basic building block is the component, and applications are built as component compositions. 'Dynamic reconfiguration' in such systems is defined as the ability to replace individual components at runtime, or to change the compositional topology by adding/removing components and/or changing the patterns of their interconnection. A 'quiescence service' supports dynamic reconfiguration by pushing a system into a stable state in which such changes can be safely made. It is thus a key enabler of dynamic reconfiguration. In this paper we present the design of a quiescence service which we have implemented for our OpenCOM component-based programming platform. We argue that our design shows significant advances over the state of the art in quiescence support for component-based systems. In particular, it minimises programmer overhead and system disruption, and efficiently supports a large class of systems without imposing architectural constraints.*

## 1. Introduction

*Dynamic reconfiguration* refers to the ability to change the functionality of an application at runtime. It is especially needed in non-stop systems (e.g. business-critical systems) so that an application can be upgraded while it is running. Another important class of system that often requires dynamic reconfiguration is the class of adaptive systems which change their behaviour in response to changes in their environment [11].

In *component-based* software systems, applications are built as compositions of components, and application structure is expressed as an architectural graph in which nodes represent components, and arcs represent connections between components. In such systems dynamic reconfiguration means not only replacing individual components at runtime, but potentially also changing application structure by adding/removing components

and/or changing the patterns of their interconnection. Several component models, including our OpenCOM model [1], support dynamic reconfiguration through an API that allows both components and connections to be dynamically created and destroyed.

However, it is not enough merely to provide a mechanism for the manipulation of component graphs. Dynamic reconfiguration is challenging because the *safety and integrity* of the system must be preserved across reconfiguration operations—and such a capability is often not provided natively in existing component systems. Preserving safety and integrity requires that the system be placed in a well-defined *quiescent state* [2] before reconfiguration can take place. But achieving quiescence is a difficult task in the general case.

In this paper we propose a generic means of efficiently achieving quiescence in component-based systems and report on our implementation of the resulting quiescence service. The goals of our approach to achieving quiescence are: (i) to avoid placing undue restrictions on the architecture and functionality of dynamically reconfigurable applications; (ii) to impose minimal work and complexity on the application programmer; (iii) to incur minimal application disturbance while dynamic reconfiguration is taking place; and (iv) to run with an acceptably low performance overhead.

The remainder of the paper is structured as follows. Section 2 clarifies the basic issues involved in supporting safe dynamic reconfiguration, and Section 3 surveys the state of the art in the area. Section 4 then briefly outlines our OpenCOM component model which forms the context of our implementation work. Subsequently, Section 5 presents our approach, Section 6 presents some implementation details and an application case-study, and Section 7 provides an evaluation. Finally, we offer our conclusions in Section 8.

## 2. Issues in dynamic reconfiguration

The key problem in dynamic reconfiguration is to determine when it is safe to perform it. We cannot simply remove or replace a component at any time: what if it is currently serving calls; or if another component was just about to call it? Therefore, we must either wait for to-be-reconfigured components to reach a quiescent state in which neither of these cases pertains, or actually force them into such a state. Following [2] we define quiescence for a component C as a state in which C is both *frozen* (i.e. currently has no threads executing in it) and *consistent* (i.e. does not contain the results of partially-completed transactions).

Depending on the characteristics and complexity of the to-be-reconfigured system, reaching quiescence can be quite challenging. The following is a list of the key issues:

- Are ‘cycle calls’ permitted?

- Are ‘blocking calls’ permitted?
- Is the system multi-threaded?
- If so, can components dynamically create ‘internal’ threads?

*Cycle calls* are calls that include the same component more than once in the call-chain. An example is given in the bottom part of Figure 1 in which Component 1 receives a call as a direct result of a call that it itself previously issued. Cycle calls lead to problems in reaching quiescence because if we take a naïve approach of simply blocking all calls to a component and waiting for it to reach a quiescent state (i.e. wait for all currently outstanding transactions to complete) we might easily induce a deadlock situation in which a cycle call never finishes because it is blocked at the component itself—in such circumstances the component will never reach quiescence.

*Blocking calls* (sometimes referred to as ‘constrained’ calls) are calls in which application code in the target component might under some circumstances block the calling thread. A blocking call would occur, for example, in a bounded buffer scenario (again, see Figure 1) in which a Producer component calls a full Buffer. A component that is currently blocking a call cannot necessarily be marked as quiescent because (as per the above definition) it may not be in a consistent state if the blocked call represents a partially-completed transaction. Furthermore, blocking calls can lead to deadlock in a similar way to that just described for cycle calls. For example, if we try to quiesce our Buffer component while it is blocking a call from the Producer, we would prevent the only event—i.e. a subsequent call from the Consumer—that could ever unblock the Producer’s call.

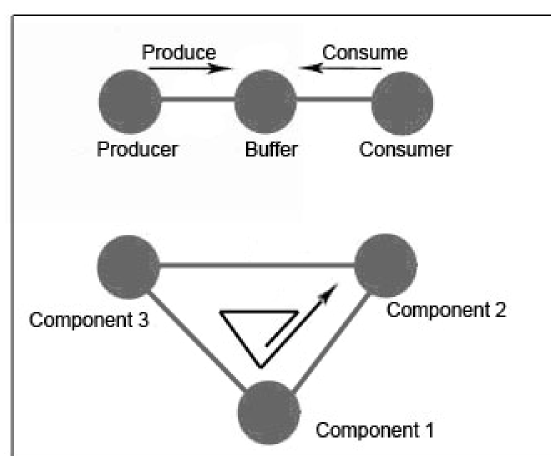


*Figure 1. Call examples*

Turning now to threading issues, we note first that it is much easier to achieve quiescence in *single-threaded* systems. This is mainly because these are inherently only affected by the cycle call problem but not by the

blocking call problem (because in such systems the single thread cannot by definition be blocked inside a to-be-quiesced component when a quiescence request is being issued). Multi-threaded systems, however, are affected by both problems (cycle calls and blocking calls) and are also prone to further complications due to race conditions.

Further thread-related problems arise if a system supports so-called *internal threads* (a thread is 'internal' to a given component if it was created by that component). Internal threads are problematic because their existence cannot be determined from 'outside' the component. Thus, even when all externally-sourced calls on a component have been observed to return, it still may not be safe to quiesce the component if there is a possibility that it contains currently-active internal threads.

## 3. Existing dynamic reconfiguration approaches

We now analyse the quiescence mechanisms employed by some existing dynamically reconfigurable systems. First, it should be mentioned that the wider context of our work is the recent general thrust towards adaptive, autonomic systems, and self-healing systems (see, e.g., the IEEE international conference series on Autonomic Computing [16]). However, this is an enormous field that, in terms of detail, is only peripherally relevant to an evaluation of our approach. Therefore we focus here on more specifically related work.

In this vein, *Conic* [2,3] was one of the first dynamic reconfiguration approaches to be described in the literature. During reconfiguration, Conic determines which components it needs to quiesce, and sends them 'passivate' requests. Then, after reconfiguration has taken place, it sends 'activate' requests to previously-quiesced components so that the application can resume normal operation. Conic works with multithreaded applications and does not impose any major architectural restrictions except that interactions between components are assumed to finish within a bounded time. It therefore accommodates a large range of applications. However, it shifts a lot of the reconfiguration-related complexity to the application level, as 'activate' and 'passivate' requests must be both initiated and handled by the application programmer. This is not a trivial undertaking (e.g. see Section 7.1) and it is also easy to make errors which will compromise safe reconfiguration.

Based on the above, Kramer and Gourdazi later proposed a dynamic reconfiguration approach that alters the trade-off—it simplifies the application programmer's job but at the same time imposes strong architectural restrictions on the types of systems that can be built. In particular, the approach applies only to 'non-cyclic, non-interleaving' systems [4]; these are single-threaded systems in which cycle calls are not allowed and only one call can be served by the system at any given time. In a similar vein, [14] describes an algorithm which achieves

quiescence in non-cyclic systems where components cannot create internal threads. It works by progressively blocking all connections between the target component and all others that can (transitively) call it.

Like Conic, the design in [13] follows a model of quiescence that waits for pending calls to finish, and at the same time potentially blocks new calls. But in this case, cycle calls or calls from other components that need to quiesce are not blocked. Similarly, [17] describes an approach for replacing/migrating remote objects in Java RMI that blocks incoming requests and waits for pending requests to finish. The algorithm deals with cycle calls but only handles object replacement (not addition or removal).

The *K42 Operating System* [9,7] from IBM supports dynamic reconfiguration using a so-called 'Read-Copy Update' approach [8]. It employs a two-phase algorithm: (i) it waits for all current calls to the target component to finish, at the same time allowing new calls made to the component and tracking them; (ii) once the current calls finish, it blocks all new calls to the component and waits for calls that started between (i) and (ii) to finish; at this point quiescence has been achieved. K42 only accommodates constrained multithreaded systems in which no 'topological' change can be made to the component graph, and where threads are guaranteed to have short lifetimes. Again, it essentially pays for a reduction in application programmer involvement in the quiescence process by imposing restrictions on the architecture of the target application and the scope of reconfiguration.

In *Polylith* [5,6], components can be replaced while they are still working—therefore the concept of quiescence is not really applicable. But once again, a trade-off is made between reconfiguration flexibility and application programmer involvement. In this case, as with Conic, flexibility comes at the expense of high programmer involvement.

Finally, prior work at Lancaster, [15] discussed requirements for a reconfiguration framework but did not include a detailed quiescence design; as such it is complementary to the present paper.

Overall, from our analysis of the above approaches we conclude that quiescence mechanisms for systems with minimal architectural restrictions typically impose a significant burden on the application programmer; whereas in systems where application programmer overhead is less, the range of applications accommodated is limited, and/or significant architectural restrictions are imposed. The quiescence mechanism to be presented in Section 5 tries to support a wide range of applications with minimal programmer overhead. It may be applied in multi-threaded systems with no architectural restrictions.

## 4. Background on OpenCOM

OpenCOM [1] is a programming language-independent component-based programming system. Each OpenCOM component 'provides' functionality through *interfaces* and 'requires' functionality through *receptacles*. Interfaces and receptacles are specified in a programming language independent interface definition language (OMG IDL), and components may bear any number of interfaces and receptacles. To enable cross-component communication, a *connection* is created between a receptacle and an interface. Connections can be created by third-parties as well as by the components actually involved in the connection. As shown in Figure 2, connections are explicitly represented as components called *connectors*, and an extensible range of connector types is available. One inbuilt connector type supports the insertion of so-called *interceptors*. These are units of code (actually, interceptors are themselves components) that are executed whenever a call is made across the connector. (As will be discussed in detail in Section 5, interceptor-capable connectors are exploited in our quiescence solution as a convenient mechanism to transparently block calls directed at a component interface.)

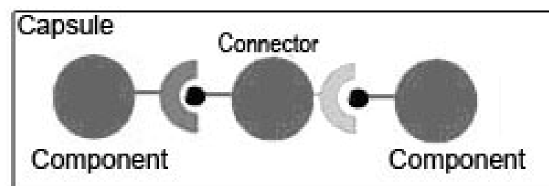


*Figure 2. Connection of components in OpenCOM*

OpenCOM components are deployed in a runtime environment which provides API functionality for creating, deleting, connecting, and disconnecting components. Therefore the basis of dynamic reconfiguration is already provided in OpenCOM—albeit without any inherent support for *safe* reconfiguration. In addition, a number of auxiliary services are provided. Two such services that are especially relied on by our quiescence service are as follows:

- the *meta-data service* allows the association at runtime of arbitrary <*name*, *value*> meta-data with components, receptacles, interfaces, operations or connectors.
- the *reflection service* enables information on the current configuration of the capsule to be obtained at run-time; this includes details of the currently running components, their interfaces and operations, and their current connections.

In OpenCOM, concurrency is orthogonal to structure and so components may freely create threads. As well as supporting multiple threads and dynamically created threads, OpenCOM freely allows thread blocking, cycle calls and internal threads.

## 5. Achieving quiescence

### 5.1 Overall architecture

Our overall dynamic reconfiguration approach employs a *two-level* architecture. The *top level* provides an application programming interface (API) for dynamic reconfiguration. This is currently provided by a system called Plastik[1] which uses an Architecture Definition Language (ADL) to define desired component configurations and also to specify how configurations may be changed at runtime [10]. However, other top level subsystems are possible: all we assume about the top level is that it conforms to an interface provided by the *bottom level* which consists of the following three generic operations:

- *ADD* – which loads, instantiates and connects a new component;
- *REMOVE* – which removes a target component; and
- *REPLACE* – which replaces a target component with another.

The latter two operations both require the target component to be quiesced before they can proceed[2]. Therefore, the bottom level of the architecture needs to provide the necessary 'quiescence service' to underpin *REMOVE* and *REPLACE*. This quiescence service is the focus of the remainder of this paper.

### 5.2 The Quiescence Service's programming model

The quiescence service is a self-contained service that is used exclusively by the bottom layer of our reconfiguration architecture as defined above (i.e. it is never accessed directly by the top layer or by applications). Its interface consists of two operations as follows:

*status = quiesce(target_component_id, timeout);*

*status = resume(target_component_id);*

*Quiesce()* attempts to quiesce the specified target component within the specified timeout, blocking until either quiescence has been achieved (in which case *success* is returned) or until the timeout expires (in which case *error* is returned)[3]. *Resume()* is used to restart a component that had previously been quiesced. It can also be called on a new component that has been installed as a result of an *ADD* or *REPLACE* call from the top level.

To work correctly, the quiescence service imposes the following *three* obligations on the OpenCOM application programmer:

---

[1] Plastik, and indeed this top architectural level, are not discussed further in this paper. Further details on Plastik can be found in the literature [10].

[2] In addition, *REPLACE* needs to capture state from the to-be-removed component and plug this into the replacement. We do not, however, consider this issue in this paper as we view transferring state as logically separate from achieving quiescence.

[3] As with other systems that impose minimal architectural constraint on programmers (see Section 3), our framework does not guarantee that a given component can be quiesced within any given time frame. This is because the framework is essentially at the mercy of application code. This will become clearer in the ensuing discussion and in Section 6.2.

1. The programmer should ensure that all OpenCOM connectors (or at least those connectors that are adjacent to components that might be quiesced) are configured with a special type of interceptor that forms an integral part of the quiescence service (this obligation can be straightforwardly met in practice by making this connector type the 'default' connector type).
2. The programmer should 'tag' all operations in the IDL interfaces of potentially-quiescable components as either *BLOCKING*, *UNBLOCKING* or *NEUTRAL*. BLOCKING is used when the associated operation blocks, or *might* block, the calling thread by putting it to sleep (note that calling another interface/operation that is itself tagged as BLOCKING does *not* qualify as blocking). UNBLOCKING is used when an operation unblocks, or *might* unblock, another operation call or calls that have previously blocked. The NEUTRAL tag is used for operations that will never block and never unblock other calls. The programmer is encouraged to avoid writing ambiguous operations which could be characterised as either BLOCKING or UNBLOCKING (e.g. an operation that blocks in one branch of an 'if' statement but unblocks some other operation in the other branch). Sometimes, however this is not easy and may even be possible; this situation is discussed in Section 5.4.1.
3. All potentially-quiescable components should implement a standard management interface as follows:

```
interface QSCallback {
    status stop_internal_threads();
    ...
}
```

   The programmer's implementation of *stop_internal_threads()* should stop all threads that have been created by the target component. If the component has not created any threads, the *stop_internal_threads()* implementation can be null and directly return a status value of *success*. When it *has* created internal threads, the complexity of the *stop_internal_threads()* implementation varies according to whether the thread(s) remain within the scope of the creating component or stray beyond it (i.e. by calling the interfaces of other components). The first case is usually straightforward as the component maintains control and visibility of the threads. The second case, however, is more tricky as such threads might block at any arbitrary point in the system (i.e. in any other component). In such a case, a *stop_internal_threads()* implementation might be obliged to return *failure* (or alternatively it might simply wait for the threads finish their operations if that occurs, and then return success).

To ease this third obligation we encourage programmers to adopt the simple *design pattern* illustrated in Figure 3: If a system is designed in such a way that a component A needs to create threads that stray beyond its boundaries, the design is re-factored to add a new component B, the sole purpose of which is to create these

threads and then have them call an operation of A's to give A control of the threads[4]. Because A did not create the threads it only needs to provide a null *stop_internal_threads()* implementation; furthermore, as it is assumed that it will never need to be quiesced, then B (as per the stipulation above) does not need to provide a *stop_internal_threads()* implementation at all. The essential purpose of this pattern is to make the component take explicitly into account any internal threads that make calls outside the component's boundary, and at the same time move thread management for the specific component away from the associated application functionality.

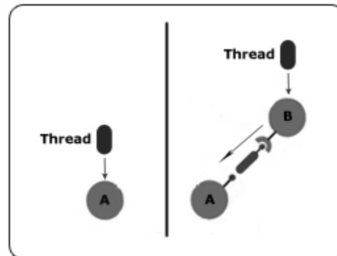


*Figure 3. The thread re-factoring pattern*

**5.3 Outline operation of the Quiescence Service**

We now outline the operation of the quiescence service, reserving discussions of certain complicating factors until section 5.4. The essence of our approach is to use the above-mentioned 'interceptor-capable connectors' that are available in OpenCOM (see section 4) to selectively block application-level cross-connector calls, so that the component on the 'interface' side of the associated connector (which is the component we want to quiesce) can move towards quiescence as extant calls on it eventually complete and return. When this process has completed successfully, the quiescence service then calls *stop_internal_threads()*on this component to stop any internal activity. And when/if *stop_internal_threads()* returns *success*, we can report back to the caller of *quiesce()* that quiescence has been achieved

Crucially, though, interceptors don't block *all* cross-connector calls: they let through cycle calls and also calls whose target operations have been tagged as *UNBLOCKING*. The former case is on the grounds that these calls must complete in order to allow the completion of calls previously made by the target component itself. The latter case is on the grounds that these calls will not themselves block, but may unblock extant calls that may currently be blocked within the component.

To manage the handling of such calls, interceptors employ a *state machine* (that supports *sm_getstate()* and *sm_update()* operations as discussed below) which dynamically determines which calls should be blocked and

[4] Wherever a component A creates straying threads it is always possible to apply this design pattern and thus reap its benefits. There is a constraint, however, that B must not contain any state. This is to ensure that if we want to remove A after it reaches a quiescent state we can safely destroy B along with the connector that connects B to A.

which should be let through the connector. There is one state machine associated with each component, and this is managed collectively by all the interceptors whose associated connectors are attached to the interfaces of the component.

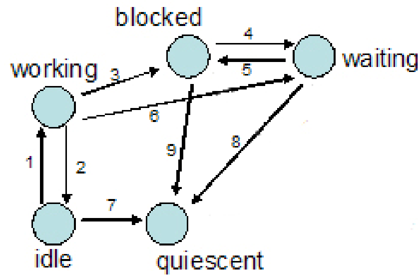


*Figure 4. State transitions undertaken by a to-be-quiesced component*

| | |
|---|---|
| *1* | *Receives call* |
| *2* | *Finishes serving all outstanding calls* |
| *3* | *Told to quiesce; serving BLOCKING calls* |
| *4* | *Finishes serving BLOCKING calls* |
| *5* | *Receives a BLOCKING call* |
| *6* | *Told to quiesce; not serving BLOCKING calls* |
| *7* | *Told to quiesce; all extant calls have completed* |
| *8* | *All extant calls have completed* |
| *9* | *All extant calls have completed* |

*Table 1. State transitions undertaken by a to-be-quiesced component*

The state machine is illustrated in Figure 4, and its transitions are described in the associated table. Essentially, while the target component has not received a ‘quiesce’ command (which is issued by the quiescence service as a result of a prior call of *quiesce()*; see Section 5.4.3), the state machine alternates between the two ‘non-quiescing’ states *working* and *idle*. On receipt of a ‘quiesce’ command (transitions 3 or 7), it moves into the realm of the ‘quiescing’ states—either directly into the *quiescent* state (following which *quiesce()* can immediately report success ), or into one of the *blocked* or *waiting* states: it transits to the *blocked* state if the associated component is currently serving calls that have been tagged as BLOCKING; and to the *waiting* state otherwise. From the *blocked* state it can transit to the *quiescent* state as soon as the component has finished serving all currently active calls. However, if the component merely finishes serving all currently active BLOCKING calls, it transits to the *waiting* state; from here, it can either return to the *blocked* state if another BLOCKING call arrives, or transit to the *quiescent* state if the component finishes serving all currently outstanding calls. The transit from *waiting* to *blocked* also covers cases in which a blocking call happens also to be a cycle call. In such cases, despite the fact that the state machine transits to the blocked state, the interceptor allows the call to pass (blocking a cycle call isn’t a viable option as the component would never quiesce).

Note that the whole of the above-described machinery is transparent to the to-be-quiesced component: the state machine manages the component's state, and the interceptors either allow or block calls to the component until it is either driven into quiescence or the timeout specified to *quiesce()* expires. A corollary of this design approach is that it is conceptually straightforward to extend our quiescence service to a distributed environment given the availability of distributed connectors (which are, in fact, provided in the OpenCOM environment [12]).

### 5.4 Discussion of complicating factors

#### 5.4.1 Ambiguous interface tagging

As pointed out in Section 5.2 it is not always easy or possible to tag a given operation unambiguously as either BLOCKING or UNBLOCKING, or to refactor operations so that a clean distinction between the two cases can be maintained. For example, a classical *consume()* operation on a bounded buffer could, depending on the situation, either block a call (i.e. when the buffer is empty) or unblock another call (i.e. when *produce()* is blocked because the buffer is currently full). In such a case *either* tag might apparently be used according to the heuristics given in Section 5.2.

In the presence of such ambiguity, calls to *quiesce()* may sometimes fail when the circumstances are unfavourable. For example, if in the above scenario *consume()* was tagged as BLOCKING and *produce()* was tagged as UNBLOCKING, failure could occur if *quiesce()* was called when the buffer was full (however, a subsequent call of *quiesce()* would succeed if the buffer had emptied by then). The 'best' characterisation thus changes dynamically depending on the present run-time state of the application.

To address such cases, we offer *dynamic tagging* as an alternative to static tagging; this is discussed in Section 5.5.2. Note, however, that even if ambiguous (static) tagging is resorted to, our approach fares no worse than other 'non architecturally constrained' quiescence designs (e.g. the relevant systems considered in Section 3 above [2,13,17]) which have no notion of tagging and are therefore completely in the dark about what operations might or might not block or unblock others[5].

#### 5.4.2 Atomic quiescence of *sets* of components

The outline operation described in Section 5.3 implicitly makes the simplifying assumption that there is only ever *one* target component to be quiesced at a time. In fact, it is often the case that *sets* of components need to be quiesced simultaneously. For example, when executing a *REMOVE* operation, not only the target component itself

[5] In any case, whether in our design or in other non-architecturally-constrained designs, it is always application-dependent whether or not quiescence can be achieved in a given application state. This a simple consequence of the fact that a key unblocking call may not be issued by the application within the required time frame.

should be removed but also the connectors attached to that component's interfaces[6]; and to remove these connectors, the components on the receptacle side must also be quiesced.

We deal with such cases by supporting a variant of *quiesce()* that takes as its argument a *set* of target components (called a *QSet*). In figure 5 and table 2 we present the additional state machinery required to cover this case (transitions 1 through 9 are the same as in figure 4). We also introduce a new state called *semi-quiescent*.

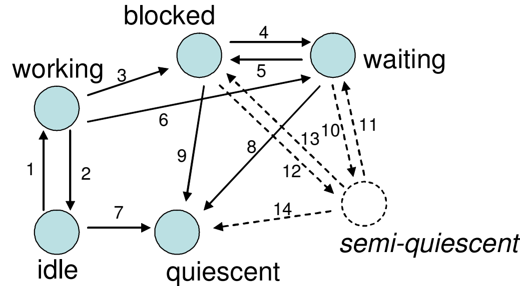


*Figure 5. State transitions undertaken when quiescing a set of components*

| | |
|---|---|
| *10* | *All extant calls have completed* |
| *11* | *Receives a call other than a BLOCKING call* |
| *12* | *All extant calls have completed* |
| *13* | *Receives a BLOCKING call* |
| *14* | *All other QSet members are semi-quiescent* |

*Table 2. State transitions undertaken when quiescing a set of components*

The intuition is that we should not block calls that originate from the other components in the QSet, because doing so would cause them never to reach quiescence. Essentially, we generalise the notion of 'cycle call' to additionally encompass calls in which *one or more of the components contained in the QSet are present in the current call chain*.

To support the required behaviour, the specified QSet is sent by *quiesce()* to all the interceptors attached to all QSet members. Each component enters the *semi-quiescent* state when all its own calls have completed, and in this state it continues to service 'cycle calls' from other QSet members (as per the above generalisation of the cycle call principle). Whenever a component enters the *semi-quiescent* state it queries all the other QSet members and, if they are all in the *semi-quiescent* state, it enters the *quiescent* state (along with the rest).

**5.4.3 Concurrency control in state machines and interceptors**

As each per-component state machine may be concurrently accessed by multiple threads executing multiple interceptors (i.e. one for each of the component's bound interfaces), concurrency control is clearly required to avoid inconsistencies as threads in interceptors read, act on, and update the state machine. However, it is crucial

[6] The reason it is necessary to remove the connectors is to avoid a situation in which a component attached to the receptacle side of the connector could make a call over a 'dangling' connection.

that this has as small an effect on performance as possible. To clarify the concurrency control issue, we reproduce here an pseudo-code rendering of the interceptor algorithm:

```
start:
  On receiving an incoming call {
    cycle_call = true if a cycle call;
    call_tag = set to either BLOCKING,
      UNBLOCKING or NEUTRAL;
    goto prepare_call;
  }
  On receiving quiesce command {
    component_is_quiescing = true;
    return;
  }
  On receiving a resume command {
    component_is_quiescing = false;
    unblock threads sleeping in prepare_call;
    return;
  }

prepare_call:
  obtain lock on state machine;
  state = sm_getstate();

  If (component_is_quiescing) {
    If (cycle_call) {
      goto do_call;
    } Else {
      If (call_tag == UNBLOCKING) {
        If (state == blocked) {
         goto do_call;
      } Else { // not blocked
        release lock on state machine;
        sleep();
        goto prepare_call;
      }
  } Else { // not cycle call or UNBLOCKING
    release lock on state machine;
    sleep();
    goto prepare_call;
  }
} Else { // component is not quiescing
  goto do_call;
}

do_call:
  sm_update(BEGIN, call_tag);
  release lock on state machine;
  call the target operation;
  obtain lock on state machine;
  sm_update(END, call_tag);
  release lock on state machine;
  return;
```

As can be seen, three kinds of incoming events are dealt with at the top of the algorithm (at *start:*):

- application-level calls on the component that have been issued from the component that owns the receptacle at the receptacle-end of the interceptor's associated connector;
- *quiesce* commands from the quiescence service requesting that the target component be quiesced; and
- *resume* commands from the quiescence service requesting the resumption of the component by unblocking threads that have been blocked in the interceptor.

The rest of the algorithm, from *prepare_call:* onwards, focuses on the handling of these events. The essential behaviour of *prepare_call:* is, as outlined in section 5.3, to let an application-level call directly though to the target component if the latter is not quiescing (i.e. if *component_is_quiescing* is false); or to let the call through even if the latter *is* already quiescing, as long as the call is a cycle call or the component is in a *blocked* state and the target operation has been tagged as *UNBLOCKING*. Otherwise, the calling thread sleeps until a corresponding *resume* event has been received. The *sm_getstate()* call simply returns the current state of the component and *sm_update()* performs the state transitions defined in Figures 4 and 5.

We turn now to the relevant concurrency control issues. Because the interceptor bases its forwarding decisions on the current state of the component, it must obtain a lock for the component's state machine to prevent the component changing state while the interceptor is making its decision. However, the interceptor releases the lock prior to executing the call on the component to enable concurrency within the component. As far as overhead of locking is concerned, we can see that a lock is obtained at the start of *prepare_call:* and released either within *prepare_call:* itself (at one of two possible locations), or within *do_call:*. And the lock is additionally obtained and released within *do_call:* itself. It can be seen that in all cases the lock is held for a very short time—in fact the time is so short that a low-overhead spin lock is the most appropriate implementation. Therefore, the inherent overhead of locking is extremely low.

The above discussion pertains to the inherent overhead of locking when an application-level call is being made from one component to another. But in addition, we need to obtain the lock within *quiesce()* when we are actually initiating the quiescence procedure. The pseudocode to do this is as follows:

```
quiesce:
  obtain lock on the target component's state
    machine;
  send quiesce command to all interceptors
    connected to the target's receptacles;
  send quiesce command to the target component's state machine;
  release lock on state machine;
  wait to be notified by the state machine
    that it has entered the quiescent state,
    or timeout;
  return success if we entered the quiescent
    state; or error otherwise;
```

Clearly, because it is only incurred 'occasionally' (i.e. when we want to quiesce a component), the locking overhead here is not as critical as it is in the case of handling application-level calls. Locking the state machine, and sending the *quiesce* command firstly to interceptors and then to the target component's state machine, deals with potential race conditions in cases where a call is about to be forwarded to the component by an interceptor.

**5.5 Additional optional features**

While the above sections set out the major features of our design, there are two additional aspects that are worthy of mention: an optional mechanism to more aggressively push a component into quiescence; and an optional mechanism to support dynamic tagging of operations.

**5.5.1 An aggressive variant**

In parallel with the quiescing process described above, the quiescence service offers the option to scan *all* components (more precisely, the state machines of all components) to see if they are currently blocking a call that has been issued from the target component: If it detects such components, the quiescence service asks them to quiesce; figure 6 illustrates the mechanism.

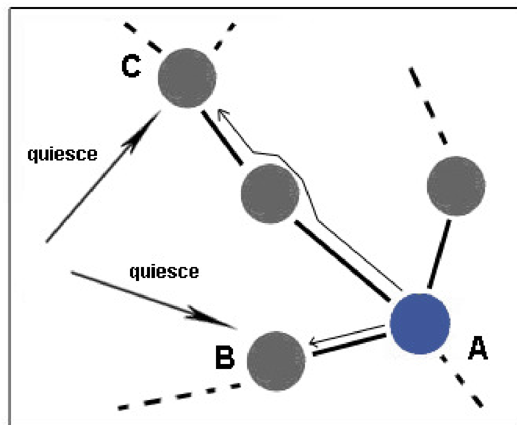


Figure 6. Aggressive quiescence mechanism

In the figure, component A is the target of a *quiesce()* call, but B and C have blocked calls emanating from A (either directly or indirectly), thus preventing it from reaching quiescence. In these circumstances, our aggressive variant will additionally instruct B and C to quiesce, which will have the effect of causing A's blocked calls to be unblocked, which in turn facilitates A reaching quiescence by helping its operations return. Note that it is not necessarily the case that B and C have to be fully quiesced for this to happen.

This optional mechanism can lead to faster quiescence but, on the other hand, it generally results in wider disturbance to areas of the application that may be 'far away' from the target of the initial *quiesce()* call. Our current default strategy for using this mechanism is to apply it is as a fall back position when an initial attempt to achieve quiescence without it has timed out.

**5.5.2 Dynamic interface tagging**

As discussed in Section 5.4.1, it is in some cases difficult to characterise interfaces unambiguously as either BLOCKING or UNBLOCKING. In such cases, the programmer can choose to tag operations *dynamically*. To achieve this, the programmer implements, for each relevant operation, an operation whose signature, *Tag*

*get_tag(Interface o, Operation o)*, is defined in the *QSCallback* interface that was introduced in Section 5.2. The quiescence service then calls these *get_tag()* implementations whenever it needs to obtain the (dynamic) tag of a given operation. Internally, *get_tag()* uses OpenCOM's reflection service to identify the target interface and operation. An example of the programmatic use of dynamic tagging is given in Section 6.2.

Clearly dynamic tagging imposes some additional burden on the programmer. However, given that it is straightforward for the programmer to determine the appropriate tagging of the operations (by knowing under what situations threads are put to sleep or woken) the overhead seems to be acceptable in the limited set of cases in which dynamic tagging may be required.

## 6. Implementation

### 6.1 Implementation details

We have implemented our quiescence service using the Java version of OpenCOM (available from http://gridkit.sourceforge.net). This version of OpenCOM implements connectors in terms of Java dynamic proxies, and dynamic component loading is based on the Java class loader. The quiescence service is implemented as a standard application-level OpenCOM component—this means that quiescence is packaged as an optional service rather than as a built-in part of the OpenCOM runtime.

The implementation of the service is considerably eased by the availability of OpenCOM's generic services (see Section 4). In particular, the meta-data service is used to record the (static) tagging of operations (as *BLOCKING* etc.). Also, when connectors are instantiated, the reflection service is used to see if a state machine has already been instantiated for associated component (and, if not, they create one). Similarly, reflection is used to determine whether or not components implement *stop_internal_threads()*. Also, OpenCOM's support for configuring default connector factories makes it straightforward to ensure that all receptacle-interface bindings employ the appropriate connector type (i.e. with the quiescence service's interceptor implementation).

One problem that arises from the use of Java relates to the detection of cycle calls. At first glance it would seem that cycles could be straightforwardly detected by examining the Java call stack at runtime. However, Java stacks do not distinguish between multiple instances of the same class, so it is possible to detect apparent cycles where in fact no cycle exists (because a different instance was involved). Mischaracterising cycle calls is a serious error, as it might cause us to 'let through' a call to an ostensibly quiescent component. One simple way to avoid this problem is to compile all components to different classes (this is our current workaround), but an alternative approach could be to provide interceptors with a stack data structure and manually add/remove calls to/from this at the beginning/ending of calls. Such an implementation would, of course, increase overhead. The essential point,

though, is that a range of means of detecting cycle calls is possible and that this issue is orthogonal to our basic quiescence design.

### 6.2 Application case-study

In this section we clarify the implementation and operation of the quiescence service by presenting an application case-study. This is based on a bounded buffer scenario like the one presented in Figure 1; the configuration, expressed in terms of OpenCOM components, is presented in Figure 7. The goal in the scenario is to replace the Buffer component using the reconfiguration service's REPLACE operation.

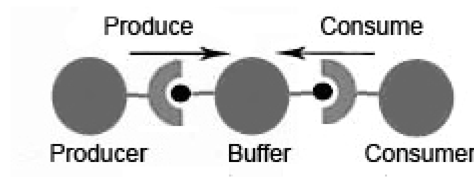


Figure 7. Case study component configuration

#### 6.2.1 The basic scenario

The *Buffer* component implements two interfaces, *IConsume* and *IProduce*, that respectively define *consume()* and *produce()* operations (*produce()* inserts an item into the buffer, and *consume()* consumes an item from it). If the buffer is empty, *consume()* goes to sleep waiting for an item to arrive (via *produce()*); and if the buffer is full, *produce()* goes to sleep waiting until an item is removed (via *consume()*). The Producer and Consumer components each have one receptacle which connects to the appropriate interface of the Buffer component.

To create the bounded buffer scenario using OpenCOM we create and connect the components using the following calls on the OpenCOM runtime:

```
consumer_component = runtime.createInstance("ConsumerComponent");
producer_component = runtime.createInstance("ProducerComponent");
buffer_component = runtime.createInstance("BufferComponent");
runtime.connect(consumer_component, buffer_component, "IConsume");
runtime.connect(producer_component, buffer_component, "IProduce");
```

The first three calls instantiate the three components (the argument of *createInstance()* is the name of the target component type). The last two calls then build the topology (the first two arguments to *connect()* refer to the components to be connected, and the third refers to the interface type at which the connection should occur).

In terms of (static) operation tagging, we characterise *consume()* as BLOCKING on the grounds that it can possibly block calls, and *produce()* as UNBLOCKING as it possibly unblocks calls. Programmatically, the tagging for the Buffer component is specified as follows:

```
public void startup() {
        setInterfaceMetaData("IConsume", "consume", BLOCKING);
        setInterfaceMetaData("IProduce", "produce", UNBLOCKING);
}
```

This *startup()* operation is part of the generic OpenCOM machinery. It is implemented on a per-component type basis and called by the runtime on instantiating a component of that type. The generic *setInterfaceMetaData()* operation is used to actually tag individual operations.

Given the above configuration, the programmer achieves the required Buffer replacement by creating a new Buffer instance and calling the reconfiguration service as follows:

```
new_buffer_component = runtime.createInstance("NewBufferComponent");
boolean result = ReconfigService.REPLACE(buffer_component, new_buffer_component);
```

The reconfiguration service determines that the only component that needs to be quiesced is the Buffer component, and it therefore calls the quiescence service accordingly (e.g. as *quiesce(buffer_component, 1000)*, assuming a default timeout value of 1 second). The quiescence service as a result sends *quiesce* commands to the two connectors and *buffer_component*'s state machine to initiate the quiescence procedure as described in Section 5.4.3.

From this point on, since *consume()* is tagged as BLOCKING, the Consumer component's connector will block all calls of *consume()*; whereas the Producer component's connector (because it is tagged as UNBLOCKING) will allow all calls of *produce()* to pass through until any previously-blocked calls of *produce()* return. *Buffer_component* will then have reached the quiescence state and its state machine informs the quiescence service which in turn will inform the reconfiguration service.

### 6.2.2 Adding dynamic tagging

In the above scenario, the Buffer component would reach quiescence in case the buffer was not full at the time the *quiesce* command was issued; but if the buffer was full, quiescence might not be achieved (see Section 5.4.1) and the reconfiguration service would have to either retry by issuing a new *quiesce()* call after a suitable interval, or report failure to the top level. So this is a situation in which the dynamic tagging option set out in Section 5.5.2 is appropriate. To enable dynamic tagging, the application programmer implements the *get_tag()* operation in the Buffer component as follows:

```
public Tag get_tag(Interface i, Operation o) {
  if (i.equals("IConsume") and (o.equals("consume")
    return (buffer = full) ? UNBLOCKING : BLOCKING;
  else if (i.equals("IProduce") and (o.equals("produce")
    return (buffer = full) ? BLOCKING : UNBLOCKING;
  else return NEUTRAL;
}
```

Here, *consume()* is dynamically tagged as UNBLOCKING when the buffer is full, and BLOCKING otherwise (and the same, *mutis mutandis*, for *produce()*). Given this tagging regime, our quiescence service would successfully quiesce the Buffer component regardless of what state it was in.

## 7. Evaluation

In this section, we first, in Section 7.1, discuss how some of the systems discussed in the related work section (Section 3) would handle the case study scenario given above; then in Section 7.2 we qualitatively evaluate our design against the goals set out in Section 1; and finally we present a performance-based evaluation in Section 7.3.

### 7.1 Related Work Revisited

In this section we relate our work to some of the systems discussed in Section 3, focusing on those that offer reasonably unconstrained multithreaded programming models (i.e. [3,13,17,14,7]; the others are too far away from our design to support a meaningful comparison). For concreteness, we make this comparison in the context of the bounded buffer case study presented in Section 6.2, and consider the specific 'difficult' case in which the Buffer is full and a call of *produce()* is blocked inside it. This is representative of a wide range of cases with non-trivial inter-call dependencies between operations that 'block' and those that 'unblock'.

Conic [3] works by issuing 'passivate' requests to all components that can potentially call the target component, the essential semantic being: "continue to serve incoming calls but do not initiate new ones". Unfortunately, passivating the Consumer in our scenario disallows calls of *consume()*, thus preventing the buffer from ever reaching quiescence. However, since passivate requests are handled by the programmer, Conic does provide the basic tools to remedy this; in particular, the programmer could 'put on hold' the passivate request inside the Consumer and the Buffer under full buffer conditions. Unfortunately, though, this would impose a significant burden on the programmer: for example, to determine if it should hold a passivate request, the Consumer code would need to 'look inside' the Buffer to determine if there was a *produce()* call blocked within it. Our design might also fail to reach quiescence if static tagging were used; but dynamic tagging would enable successful quiescence under any circumstances with far less burden on the programmer (see Section 6.2.2). This is essentially because of the explicit distinction we draw between 'blocking' and 'unblocking' operations. In addition, our design would allow other components that can potentially call the target component (there aren't any in the specific scenario) to potentially continue their operation as long as they don't attempt to actually call the target.

Of the other relevant designs, [13, 17, 14] would in our scenario all block calls of *consume()* and therefore preclude any possibility of achieving quiescence. Furthermore, unlike Conic and our design, there is no way to deal with this situation by programmer intervention. The only alternative is to attempt to quiesce again at a later time, hoping that no threads are blocked inside the Buffer at that time. The two-phase algorithm employed by [7] *could* work correctly under our circumstances as the first phase would allow a *consume()* call to proceed.

However, if another ‘blocking’ *produce()* call was issued between the two phases, then the algorithm would once more block the very *consume()* call that would potentially unblock it. And again, there is no scope for programmer intervention.

In conclusion, because none of them takes application level blocking/unblocking semantics into account, none of the systems surveyed here properly addresses the difficult inter-call dependency cases discussed above. They either fail to properly address such cases, shift the complexity burden onto the programmer, or restrict the programmer with structural and functional constraints such as not allowing cycle calls, requiring threads to have a short lifetime, or preventing components from creating internal threads.

**7.2 Qualitative evaluation**

In this section we evaluate our design against the goals identified in the Introduction. The first of these was “*to avoid placing restrictions on the architecture and functionality of the dynamically reconfigurable system*”. As we have seen, our quiescence service freely allows multiple threads, cycle calls, blocking of calls at the application level, and components arbitrarily creating threads at run time. The only real architectural restrictions that remain are that reconfigurable OpenCOM systems must use a special type of connector/interceptor (as mentioned in Section 6.1, this is easily arranged using a single configuration option). Of the systems surveyed in Section 3, only Conic supports such an unconstrained programming environment but at the expense of pushing complexity onto the application developer. As discussed in Sections 3 and 7.1, all the other approaches we have examined offer a significantly constrained programming environment.

The second goal was “*to impose minimal work and complexity on the application programmer*”. As we have seen, the programmer is required to tag all operations and to implement the *stop_internal_threads()* operation on all potentially-reconfigurable components. This, however, represents a relatively modest overhead. The tagging of operations is straightforward and reasonably mechanical (even in cases where the programmer chooses to tag operations dynamically). And, as discussed in Section 5.2, the implementation of *stop_internal_threads()* is rendered straightforward through the use of the proposed thread design pattern. The programmer burden thus compares favourably with the comparable systems surveyed in Section 3.

The third goal was “*to incur minimal application disturbance while dynamic reconfiguration is taking place*”. Leaving aside the aggressive variant described in Section 5.5.1, the quiescence process in REPLACE operations directly affects only the target component to be removed, and in REMOVE operations the target plus components that have receptacles connected to the target’s interfaces. In the latter case, system disturbance is related to the topology of the application; for example in a ‘hub’ topology where all components depend on a ‘central’

component, all of the components would be affected; however in conducive systems the set of dependent components, and thus application disturbance, is correspondingly less. In both the REPLACE and REMOVE cases any part of the system that does not depend on the components to be quiesced will not be affected at all. With the 'aggressive' variant we disturb the system in a different way. Choosing whether or not to apply this option gives us some control over the trade-off between global disruption but potentially faster completion, or local disruption but potentially slower completion. However, as the amount of time needed for a call to unblock is application specific, the optimal choice of one or other of these possibilities ultimately depends on the design and current state of the system. Of the comparable systems surveyed in Section 3, all are inherently more 'aggressive' and coarse-grained than our approach and, because they do not distinguish blocking from unblocking calls, they will generally tend to quiesce more of a given system than is strictly necessary.

### 7.3 Performance overhead

The fourth goal from the introduction was "*to run with acceptably low performance overhead*". Observe, however, that the most significant performance metric for any quiescence design is *not* the time taken to achieve quiescence as this is inherently dependent on the design and current state of the particular system under consideration. Instead, the key performance metric is the intrinsic 'background' overhead incurred by the system while a *quiesce()* request is *not* in progress. As this overhead in incurred exclusively within our interceptor algorithm and state machine (see section 5.4.3), we evaluated it by comparing the following cases:

1. The overhead of inter-component calls without interceptors (i.e. direct component-to-component connection). This represents an ultimate 'baseline' of a straight component-to-component call involving only the single level of indirection that is inherent in any reconfigurable component-based system.
2. The overhead of calls across an interceptor with the state machine and interceptor algorithm enabled; this represents the full overhead of a system that is capable of being acted on by our quiescence service.

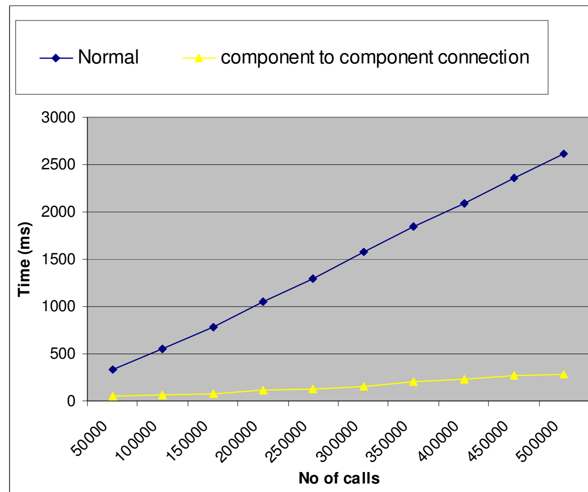


*Figure 8. Overhead of interceptor algorithm when calling a null target operation*

In each of these cases we measured the times taken to call a target operation that takes no parameters and has an empty body (averaged over a large number of calls). Figure 8 illustrates the results. As can be seen, there is an approximate overhead of 900% in the presence of our quiescence mechanism. This large overhead is due to the use of proxy objects that handle calls directed to interceptors and components in our Java implementation.

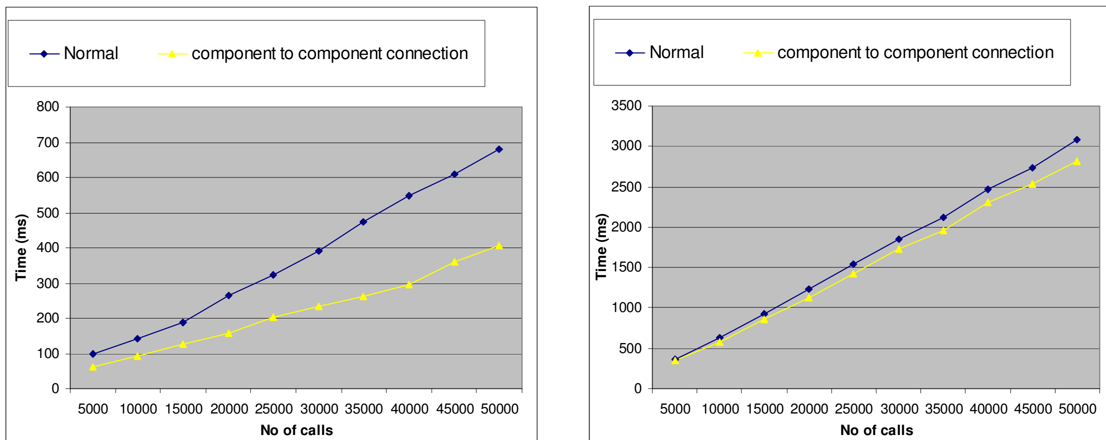


*Figure 9. Overhead with a non-null target operation (10,000 empty loop iterations at left, 50,000 at right)*

While this overhead may appears to be prohibitive, the effect is far less significant than it might appear. This is because it is fair to assume that in almost all most real applications the overhead of computation within an operation body will dwarf the overhead of calling the operation. To investigate the overheads of our quiescence service under this assumption, we repeated the above experiment but this time employing a non-null target operation which encapsulates a number of iterations around an empty-bodied 'while' loop (see Figure 9). In the first graph in Figure 9, the number of loop iterations is 10,000 and the overhead reduces to approximately 70%. In the second graph, with 50,000 iterations (still a very modest computational load), there is almost no significant

comparative overhead. We conclude from this that the empirical overhead of supporting dynamic reconfiguration using our design is likely to be negligible in the vast majority of real application scenarios.

## 8. Conclusions and future work

We have presented a quiescence service that is designed to support dynamic reconfiguration in multithreaded component-based programming environments. The service successfully addresses the goals set out in Section 1 without incurring the compromises that earlier systems have made (see sections 3 and 7). Essentially, we support an unconstrained multithreaded programming environment and impose only minimal (and well-defined) obligations on the application programmer. The explicit distinction that we draw between 'blocking' and 'unblocking' operations is a key and novel contribution, as discussed in Section 7. In terms of mechanisms, our design is based on interception in connectors and makes use of features of the OpenCOM programming model (such as the meta-data service and the reflection service). Nevertheless, it is clear that the general principles of the work can be applied equally well to other component based programming systems.

In future work we plan to address two distinct areas. First, we want to exercise our system in a wider range of realistic application scenarios. Within our OpenCOM-based GridKit platform [12] we have developed a large number of 'component frameworks'—i.e., component architectures that support dynamic reconfigurability. However, we currently rely heavily on the application programmer to achieve safe dynamic reconfiguration using ad-hoc means (e.g. per-component framework patterns). By adding a combination of the quiescence service and Plastik, we plan to re-design these component frameworks to be far simpler and more robust in the face of dynamic reconfiguration.

The second area for future work is to extend the quiescence service to support safe *distributed* reconfiguration. Again, we will use our GridKit platform as the context for this work. In principle, it should be relatively straightforward to generalise to the distributed case as the design already incorporates the notion of connectors.

## References

**[1]** Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T., "*A Generic Component Model for Building Systems Software*", to appear in ACM Transactions on Computer Systems, 2008.

**[2]** Kramer, J., Jeff Magee, J., "*The Evolving Philosophers Problem: Dynamic Change Management*", IEEE TSE 16, 11, November 1990.

**[3]** Magee, J., Kramer, J., Sloman, M., "*Constructing Distributed systems in Conic*", IEEE TSE 15, 6, June 1989.

**[4]** Gourdazi, K.M, Kramer J, "*Maintaining node consistency in the face of dynamic change*" Proc. Third International Conference on Configurable Distributed Systems, 1996. Pages: 62 – 69, 6-8 May 1996.

**[5]** Hofmeister, C., Purtilo, J., "*A Framework for Dynamic Reconfiguration of Distributed Systems*", Proceedings of the 11th International Conference on Distributed Computing Systems, Pages 560-571, 1991.

**[6]** Hofmeister, C., "*Dynamic Reconfiguration of Distributed Applications*", Phd Thesis, 1993.

**[7]** Soules, C.A.N., Appavoo, J., Hui, K., Silva, D.D., Ganger, G.R., Krieger, O., Stumm, M., Wisniewski, R.W., Auslander, M., Ostrowski, M., Rosenburg, B., Xenidis, J., "*System Support for Online Reconfiguration*", In Proc. USENIX Annual Technical Conference, June 2003.

**[8]** McKenney, P.E., Slingwine, J.D., "*Read-Copy Update: Using Execution History to Solve Concurrency Problems*," International Conference on Parallel and Distributed Computing and Systems, 28–31 October 1998.

**[9]** Appavoo, J., Auslander, M., DaSilva, D., Edelsohn, D., Krieger, O., Ostrowski, M., Rosenburg, B., Wisniewski, R.W., Xenidis, J., "*K42 Overview*", IBM TJ Watson Research, 2002.

**[10]** Joolia, A., Batista, T., Coulson, G., Tadeu A., "*Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform*", to appear Proc. 5th Working IEEE/IFIP Conference of Software Architecture (WICSA 5), Pittsburgh, Pennsylvania, USA, 6th-9th November 2005.

**[11]** Duran-Limon, H.A., Blair, G.S., Coulson, G., "*Adaptive Resource Management in Middleware: A Survey*", IEEE Distributed Systems Online, 5, 7, July 2004 (http://dsonline.computer.org).

**[12]** Blair, G., Coulson, G., Grace, P., "*Research Directions in Reflective Middleware: the Lancaster Experience*", Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004) co-located with Middleware 2004, Toronto, Ontario – Canada, pp 262-268, October 18th 2004.

**[13]** Almeida, J.P.A, Wegdam, M, Marten van Sinderen and Nieuwenhuis, L. , "*Transparent Dynamic Reconfiguration for CORBA*“, Proc. 3rd International Symposium on Distributed Objects and Applications,(DOA '01), 2001.

**[14]** Rasche, A. and Andreas, P. "*Configuration and dynamic reconfiguration of component-based applications with Microsoft .NET*", Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC'03), 2003.

**[15]** Hillman, J., Warren, I, "*An Open Framework for Dynamic Reconfiguration of Component-Based Systems*", 26th International Conference on Software Engineering (ICSE'04), 2004 .

**[16]** The IEEE International Conference Series on Autonomic Computing, www.acis.ufl.edu/~icac2008/, 2007.

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