Dynamic Module Replacement in Distributed Protocols

Nigamanth Sridhar, Scott M. Pike, and Bruce W. Weide
Computer and Information Science
Ohio State University
2015 Neil Ave
Columbus OH 43210, USA
{nsridhar,pike,weide}@cis.ohio-state.edu

Abstract

Dynamic module replacement — the ability to hot swap a component's implementation at runtime — is fundamental to supporting evolutionary change in long-lived and highly-available systems. Most existing solutions require special-purpose middleware or depend on research languages with limited support for mainstream software development. We present a language-neutral technique for dynamic module replacement using Service Facilities (Serfs) — a pattern-based design strategy for decoupling runtime dependencies. We demonstrate the sufficiency of Serfs with respect to a litmus test of criteria for module replacement. Next, we extend the traditional scope of module replacement to encompass the domain of modules for distributed protocols. We conclude by applying the Serf strategy to illustrate dynamic replacement of mutual exclusion protocols in modules for distributed resource allocation.

1 Introduction

Dynamic reconfiguration refers to changing, updating, or otherwise modifying a system during execution. Dynamic reconfiguration is essential to supporting the operation and evolution of long-lived and highly-available systems. For online systems, such as banking applications, it may be economically prohibitive to compromise system availability for either planned or unplanned downtime. For distributed systems, such as telecommunication switching networks, it may be impossible even to coordinate downtime if the application spans across multiple administrative domains. To accommodate the demands of evolutionary change, such applications must support on-the-fly system modification.

We consider a mode of dynamic reconfiguration known as *dynamic module replacement* that enables the implementation of a component to be hot swapped at runtime. Existing solutions have various shortcomings that limit their scope of applicability and deployment. For example, the approaches taken by CONIC [12], ARGUS [2], and POLYLITH [8] are coupled to particular research languages that generally lack support for mainstream software development. Other solutions have targeted development languages, but at a cost: HADAS [1] (for Java) requires integrated tool support at runtime; dynamic Java classes [13] compromise portability by modifying the Java Virtual Machine; and dynamic classes for C++ [7] lack an effective means of completing module replacement, because existing instances cannot be changed (replacement is delayed until they go out of scope). Several surveys on these and other approaches to module replacement discuss merits and drawbacks that extend beyond the scope of this paper [6, 18, 3].

Our present concern is to support module replacement with an effective, language-neutral approach that can be readily adopted and deployed by practitioners engaged in distributed software development. These desiderata call for a general strategy that can target mainstream languages without resorting to special-purpose middleware or extensions to standard runtime environments. The cornerstone of our solution is a pattern-based design strategy known as *Service Facilities* (Serfs) [20]. As a composition of well-established design patterns [4], Serfs provide a strategy for decoupling advanced dependencies in software. In particular, Serfs are sufficient for decoupling and rebinding runtime dependencies that are central to the problem of dynamic module replacement.

The rest of this paper is organized as follows. Section 2 covers the background and technical framework of Service Facilities. Section 3 demonstrates the sufficiency of Serfs with respect to a litmus test of criteria for dynamic module replacement. In Section 5, we extend the scope of dynamic replacement to support the hot swapping of modules for distributed protocols. Section 4 applies the Serf strategy to illustrate dynamic replacement of mutual exclusion protocols in modules for distributed resource allocation. We summarize our results and conclusions in Section 6.

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE

2 Background and Technical Framework

Service Facilities (or Serfs for short) are a pattern-based design strategy for decoupling advanced dependencies in software. Serfs compose elements of several established design patterns from the Gang of Four, such as Memento, Abstract Factory, and Strategy [4]. Typically, such design patterns documented successful approaches to decoupling an *individual* software dependency. Serfs were designed as a composite strategy (based on these patterns) to decouple problems compounded by *intersecting* dependencies. The Serf approach to dynamic module replacement is based primarily on the Abstract Factory and Strategy patterns. Accordingly, we introduce the pertinent aspects of Serfs in this context, and refer the reader to [20] for a more detailed treatment of Serfs and their applications.

Abstract Factory. An object is an instance of a concrete class and is created by invoking a constructor. Most OO languages (including Java, C++ and C#) couple the name of an object's constructor to the name of its concrete class. Equating constructor names with class names introduces a dependency between a client and each concrete class it uses. This complicates software maintenance, because replacing a module (even statically, prior to compile-time) requires substituting the name of the new class everywhere instances of the old class were created in the client code.

Abstract factories simplify static module replacement by decoupling client-to-class dependencies. In the factory pattern, a client instantiates a *factory* class for each *product* class it uses. To obtain a product instance, the client invokes create() on the appropriate factory. The factory, in turn, invokes the product class constructor, and returns to the client a reference to the newly created instance. Thus, although the client depends directly on factories, its dependence on product classes is decoupled by a level of *indirection*. This allows product classes to be statically replaced simply by instantiating a new factory for that product.

Factories introduce a level of indirection solely to decouple object creation; afterwards, however, the client is stuck with a client-to-product dependency which persists until the product goes out of scope. To decouple runtime dependencies, Serfs maintain the level of indirection for the duration of the product's lifetime. As such, Serfs are a logical extension of abstract factories.

How do Serfs and factories differ? When a client calls create() on a Serf, the Serf invokes the product constructor just as a factory would. Unlike factories, however, the Serf encapsulates the new product inside itself, and only gives the client a *logical* handle to the product. The logical handle has as part of its private state a reference to the actual object. These references maintain a permanent level of indirection between the client and the product objects. Henceforth, when the client wants to perform an operation on the product object, the invocation is made on the Serf, with the logical handle as a parameter. The Serf uses the logical handle to delegate the operation to the object it encapsulates. Figure 1(a) shows a typical method invocation.

Strategy. C++ and Ada provide direct language support for parameterized programming with templates and generics, respectively. For example, a component for processing bank statements could sort checks by date, or by check number. The actual ordering can be factored out of the sorting algorithm and packaged as a parameter. The binding of this parameter can be (statically) selected as late as compile-time. In languages without templates, such as Java and C#, the Strategy pattern [4] can be used to avoid coupling a sorting algorithm to a hard-coded ordering scheme. The ordering scheme can be factored out into a separate module, and passed to the sorting class constructor at runtime to bind the ordering scheme to the sorting algorithm.

Serfs extend the Strategy pattern to allow parameterized components to be instantiated at runtime. The power of the parameterization mechanism in the Serf approach is the same as the power of C++ templates. The main difference is the binding time. Before a Serf can be used to create and operate on objects, all of its parameters have to be set. In order to do this, each Serf class has methods of the form set*Parameter* corresponding to each parameter (named *Parameter*). Since the Serf approach binds parameters to templates at run-time, it also supports *rebinding* of parameters in the event that any of the decisions change.

3 Module Replacement with Serfs

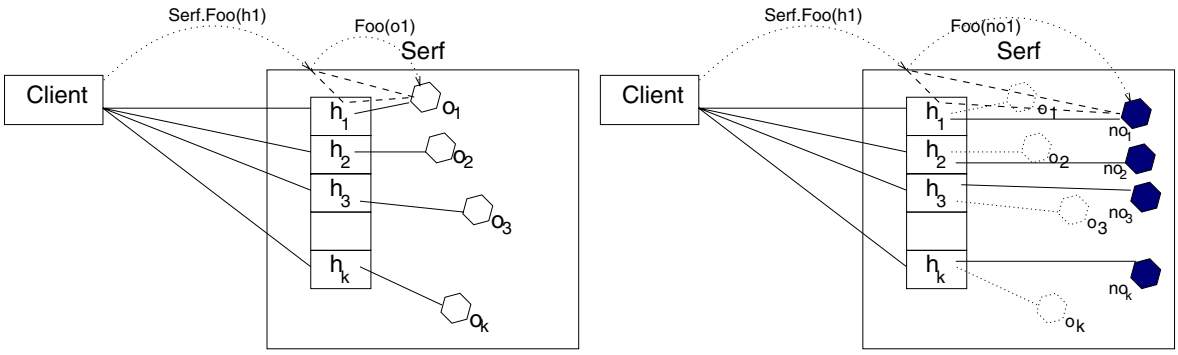
This section outlines five sufficient factors for supporting dynamic module replacement, and describes how Serfs satisfy these prerequisites. Previous research has explored other factors pertaining to the safety and semantic correctness of module replacement [9, 8]. A mature reconfiguration strategy should address these additional factors such as protection, security, and substitution consistency. Ultimately, however, all approaches must supply a sufficient infrastructure for effecting the module replacement itself. Our concern is to supply such a infrastructure using Serfs.

3.1 Requirements for Module Replacement

In a running system, what is required to substitute the implementation of a module without taking the system down? Below we present a litmus test of five steps which are sufficient for realizing dynamic module replacement.

1. **Initiation:** Module replacement must be initiated, either internally by the module itself, or externally by a third-party with access to the execution environment.

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE



(a) Serf delegating client invocation. When the client makes an invocation on its logical handle h_1, the Serf dispatches the call to the corresponding target object o_1. Subsequently, the Serf returns the invocation results to the client.

(b) After instance rebinding, the old objects are destroyed, and the client handles now point to the new object instances. A client invocation on h_1 is now dispatched to the new object instance no_1.

Figure 1. During the instance rebinding step of module replacement, the Serf creates links from the client handles to the new object instances. Once these links have been created, the old object instances are destroyed. All future method calls are redirected to the new object instances.

2. **Module Integrity:** The consistency of modules undergoing replacement must be preserved. Typically this can be achieved by restricting or regulating how client invocations can (or cannot) be interleaved with module replacement activities.
3. **Module Rebinding:** The new (target) replacement module must be dynamically loaded and linked into the runtime environment, so that new object instances can be created to replace their old counterparts.
4. **State Migration:** The abstract state of each current object instance must be externalized, transmitted, and then internalized to reconstruct an equivalent abstract state in the corresponding object instance of the target replacement module.
5. **Instance Rebinding:** Each client-side object handle must be redirected to its new object instance counterpart. This involves rebinding old object handles to new instances of the target replacement module. Old object instances should be finalized.

3.2 Realizing Module Replacement with Serfs

Now let us see how Serfs actually satisfy the five conditions presented in Section 3.1. The *initiation* and *module rebinding* steps depend on runtime environment support. *State migration* can be realized directly by the Serf, or indirectly by the original and target modules themselves. The material support offered by Serfs pertains primarily to enforcing *Module integrity* and *Instance rebinding*.

Initiation. Internal initiation is straightforward; a module triggers reconfiguration in response to a condition in its local state or environment, such as a fault, performance degradation, etc. Internal initiation is essentially *planned* change, and so has limited scope. Responding to *unplanned* change requires support for external initiation. Environments that support remote invocations, such as Java (RMI) and .NET (Remoting), enable a third-party to send control messages and invocations to running applications. Thus, a third-party can initiate module replacement by invoking set*Parameter* on a Serf, with the new target module.

Module Integrity. Access control is required when interleaving client invocations can disrupt module replacement or compromise the integrity of the object instances themselves. Since the Serf wraps the module implementation (Section 2), it serves as a "gateway" which can defer method calls while effecting module substitution. This creates a local synchronization mechanism that may delay servicing a client invocation during module replacement. After completing the reconfiguration, however, deferred invocations can be delegated to the new target objects.

Module Rebinding. The initiation step provides the Serf with information about the new module implementation. In the case of environments that support dynamic class loading, such as Java and .NET, the initiating agent can supply the name of the new implementation directly. The Serf can then locate and load the new class, and then create instances of the new module type. In the case of .NET applications, the initiator can simply point the Serf to the source code of the new implementation, which the Serf can then

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE

compile, load, and then create instances.

State Migration. Migration amounts to recording a snapshot of each object's abstract state, and then reconstructing an object of the target instance with an equivalent abstract value. This can be achieved using two complementary operations: one for externalizing an object's concrete state into the abstract value it represents, and one for internalizing an abstract value into an object by constructing a concrete state that represents it. Object instances can support their own state migration by implementing such value-transmission methods directly [5]. Alternatively, these methods can be realized indirectly by the Serf itself, provided that the original and target modules support observability and controllability of their abstract states [22].

Instance Rebinding. This is the fundamental step in module replacement. The primary task is to decouple the dependency between the module interface (known to the client), and its runtime objects (which realize that interface). Parameterization mechanisms such as C++ templates can decouple this dependency at design-time, but the problem of dynamic module replacement requires decoupling this advanced dependency *at run-time*. Essentially, each client-side object handle must be rebound to its new target instance. As a wrapper, a Serf can maintain an internal mapping of handle-to-target bindings. These bindings are updated during the state migration phase of module replacement. Subsequent client invocations are simply intercepted by the Serf, and then redirected to their new target instances. Figure 1 illustrates instance rebinding in a Serf.

Discussion. Both Java and C# both support dynamic class loading and remote method invocation, thus enabling dynamic module replacement using Serfs. Further, since all .NET languages compile to the same intermediate language (MSIL), all features available in C# are available in all .NET languages [14], thus making the Serf approach uniform across all .NET languages. We have outlined how Serfs provide a flexible, language-neutral infrastructure that can be used in mainstream, production platforms that support reflection to provide an easy approach to dynamic module replacement.

4 Implementing Mutual Exclusion using the Serf approach[1]

Previous treatments of dynamic module replacement have focused almost exclusively on containers for encapsulating data structures(such as queues, stacks, and maps) [2]. Unfortunately, this peculiar emphasis on data containers has underestimated — and perhaps mis-characterized — the full power of module replacement. The first contribution of this paper has been to illustrate the sufficiency of Serfs with respect to supporting dynamic module replacement. The second contribution will show that the limited focus on containers has been unwarranted; in fact, module replacement is as powerful as our ability to modularize various aspects of computation [16]. In this Section, we present an excursion into the landscape beyond containers to suggest how other modes of dynamic reconfiguration can be subsumed by module replacement.

The problem of *mutual exclusion* involves the synchronization of the activities of (logically) concurrent processes. The synchronization can be viewed as a way to coordinate access to a shared resource, or as a privilege to execute a particular Section of code. The problem of mutual exclusion has been well studied and numerous solutions are available in the literature. For a comprehensive presentation of mutual exclusion and applications see [10, 11]. For a list of such solutions, we refer the reader to [15].

A common drawback of most solutions to mutual exclusion is that the processes in the network are intimately tied to a particular conflict resolution protocol. Once the network application has been deployed, it is not possible to change the conflict resolution policy without stopping the processes, or using some specialized configuration languages not suitable for mainstream development environments. In the ideal case, we should be able to change the conflict resolution policy while the system layer is running normally. The problem, however, with replacing the conflict resolution layer is that clients have direct access to the representation of the conflict resolution layer. Since this is the case, there is no way of changing the protocol at runtime without the clients being aware of (and actively participating in) such a change. As a solution to this problem, we invoke the folk theorem that most problems in computer science can be solved by introducing an extra level of indirection. We separate the conflict resolution layer (which is encapsulated in a module) from the client by introducing an extra level of indirection.

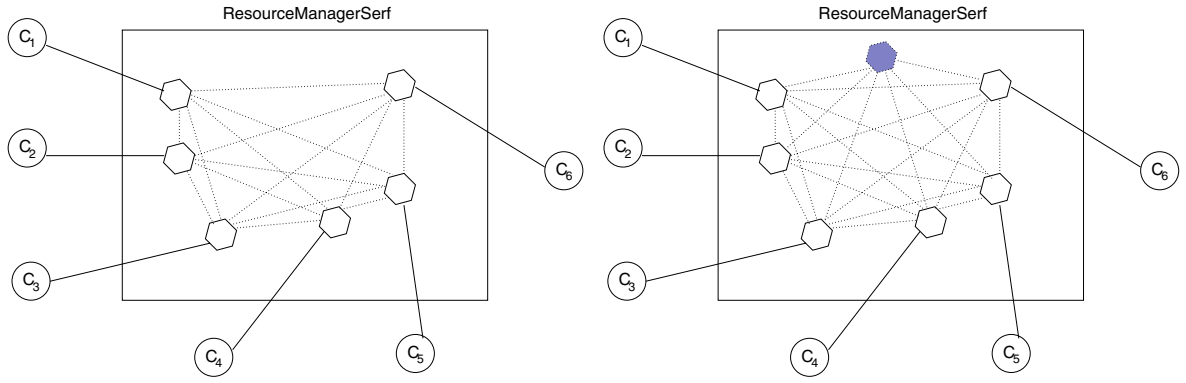
We use the Serf approach to do this in the following manner. When the system is initialized, and the resource is created, a corresponding resource manager (ResourceSerf) is created as well. Thus each resource in the system has its own resource manager which controls access to the resource according to the appropriate access policy (mutual exclusion, read-only, etc.). The access policy itself is separate from the resource, and is encapsulated in a different module (ProtocolSerf). The ResourceSerf is instantiated with a particular implementation of ProtocolSerf as template parameter. The ResourceSerf is then registered with a trading service provided by the environment[2]

When a client wants to contend for a particular resource,

[1] Due to lack of space, we have presented only the key aspects of our implementation of the resource manager Serf. We refer the interested reader to our complete experience report in developing this system [19].

[2] Examples of trading services are the CORBA Trading Service, and Microsoft and IBM's UDDI

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE



(a) Conflict graph before reconfiguration

(b) Conflict graph during reconfiguration — includes the proxy representing the Serf.

Figure 2. The conflict graph of proxies representing the various clients in the network. The second figure shows the conflict graph including the Serf's own proxy that is used to achieve the access control requirement for module replacement.

it contacts the trading service to actually acquire the reference to the appropriate ResourceSerf. Once the client has contacted the ResourceSerf, it invokes create() on the ResourceSerf to register itself. In the create() method, ResourceSerf creates a proxy object of type Resource, and wraps it in a logical handle. Then it invokes create() on the ProtocolSerf to create a protocol proxy object that represents the newly created Resource proxy in the conflict resolution layer. ResourceSerf finally hands the handle to the client. From this point on, the client views this handle as the actual resource itself.

Further, ResourceSerf continues to hold a reference to the handle. This reference is important in order to perform dynamic module replacement. All objects and handles created by the ResourceSerf therefore exist logically inside the ResourceSerf. Since the ResourceSerf is a long-running component, and may potentially never go out of scope. Since the ResourceSerf has references to all the client handles, these handles will never get garbage-collected — even if the client that created them goes out of scope. In order to avoid this potential problem of memory leaks, the Serf maintains a *weak reference* [21, 17] to the logical handles. Since this reference is only valid in the presence of some strong reference pointing to the same object, as soon as the client goes out of scope, the handle will be ready for garbage collection.

An instance of Resource is modeled by two boolean variables – requested and available. When a client requests its resource r, r.requested is set to **true**. Then when the resource is available for use by the client, r.available becomes **true**. Only one of the resource proxies in the system can have both requested and available set to **true** at the same time (thus satisfying mutual exclusion).

Upon becoming hungry (need to use resource), a client program invokes the request() method on the ResourceSerf, with its resource proxy as parameter. The Serf modifies the abstract state of the Resource proxy to record the request, and then forwards the request on to its representative in the conflict resolution layer. At this point, control returns to the client, who can perform some alternate computation while waiting for the resource to become available. The client then periodically queries the Serf to see if the resource is available by invoking the isAvailable() method, and when the resource does become available, the client starts to use it after calling get() on the Serf. When the client is done using the resource, it signals the ResourceSerf by calling the release() method.

4.1 Dynamic Reconfiguration in ResourceSerf

In this Section, we illustrate the replacement of the conflict resolution protocol that is being used to determine which client in the network currently has access to the resource.

Initiation. A console program can prepare a message and pass it to ResourceSerf to initiate the reconfiguration. In this particular case of protocol replacement, the message that must be sent is an invocation to the setProtocolSerf method (Figure 3). As parameter to this method, the initiator will pass a Serf that implements the new conflict resolution protocol. ResourceSerf can then reconfigure itself to use this new protocol instead of the old one. The entire

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE

```
public void setProtocolSerF(ProtocolSerF psf)
{
  /*———————- Segment 1 ———————-*/
  if (protSerf == null) // First time
  {  protSerf = psf;  }

  /*———————- Segment 2 ———————-*/
  else // reconfiguration
  {
    if (pMapSerf.size(rMap) == 0)
    {  protSerf = psf;  }

    /*———————- Segment 3 ———————-*/
    else
    {
      ProtocolSerF newProt = psf;
      ProtPxy serfProxy = (ProtPxy) protSerf.create();
      ProtPxy newSerfP = (ProtPxy) newProt.create();

      /*———————- Segment 4 ———————-*/
      // Acquire resource under both protocols
      protSerf.request(serfProxy);
      while (!protSerf.isAvailable(serfProxy)) {}

      newProt.request(newSerfP);
      while (!newProt.isAvailable(newSerfP)) {}

      PMap tempMap = (PMap) pMapSerf.create();
      pMapSerf.swap(tempMap, rMap);

      /*———————- Segment 5 ———————-*/
      while (pMapSerf.size(tempMap) > 0)
      {
        Resource tempR = (Resource) this.create();
        ProtPxy tempPP = (ProtPxy) protSerf.create();
        ProtPxy newTempPP = (ProtPxy) newProt.create();

        pMapSerf.undefineAny(rMap, tempR, tempPP);
        if (protSerf.isRequested(tempPP))
        {   newProt.request(newTempPP);  }
        pMapSerf.define(rMap, tempR, newTempPP);
      }

      /*———————- Segment 6 ———————-*/
      // Replace the protocol serf.
      protSerf = newProt;

      // Release the resource under the new protocol
      protSerf.release(newSerfP);
    }
  }
}
```

Figure 3. The setProtocolSerF **method in** ResourceSerf **(Centralized gateway version)**

module replacement process is handled inside this method, as detailed in the subsequent steps below.

Module Integrity. What happens if the request for reconfiguration arrives at a time when the resource is held by one of the clients in the network? What happens to the messages, if any, that are in transit at the current time? When is it safe to perform the reconfiguration, while ensuring that there are no interference effects?

In order to ensure safety, the ResourceSerf must make sure that the resource is not in use by any client in the network. To achieve this, the Serf makes itself one of the clients of the resource. It thus gets issued its own proxy, and the Serf itself requests access to the resource. When the Serf gets access to the resource, no other client process can be holding the resource, since the protocol guarantees mutually exclusive access to the resource. Therefore, changing the protocol at this time would not affect any of the clients. In fact, the clients are completely unaware of this change. Segments 3 and 4 in Figure 3 show the Serf acquiring exclusive access to the resource. Figures 2(a) and 2(b) show the ResourceSerf participating in the conflict resolution. Originally the conflict graph consists of all the client proxies. Each edge in the conflict graph represents a contention for the resource. No two neighbors in the graph can access the resource simultaneously. Since this is a fully connected graph, only one node can have the resource — thus satisfying the spec for mutual exclusion.

Module Rebinding. The module that implements the new conflict resolution protocol is created as part of the initiation step by the external initiator program. This new module is passed into the Serf through the method invocation. This step involves binding the new ProtocolSerf module to the ResourceSerf template. One of three scenarios could be true when the setProtocolSerF (Figure 3) method is invoked: (1) ResourceSerf is a fresh template, and is being instantiated for the first time. This case is trivial, and is handled in Segment 1 in the method. (2) ResourceSerf is being reconfigured, but no proxies exist in the network. This case is identical to case (1) and is handled in Segment 2. (3) ResourceSerf is being reconfigured, and there are live proxies in the network. This is the interesting case. Before this new module can be bound in Segment 6, the state migration and instance rebinding steps must be completed. Once they have been completed, then the rebinding itself can be done.

State Migration. State migration in this example consists of taking a checkpoint of the abstract states of each of the proxies under the old protocol module, and "rolling forward" the new protocol to this checkpoint. In this example, the ResourceSerf directly takes this checkpoint, by individually querying each proxy instance for its abstract state, and setting a new proxy instance to the same state. The abstract state of each proxy is a pair of booleans, one of

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE

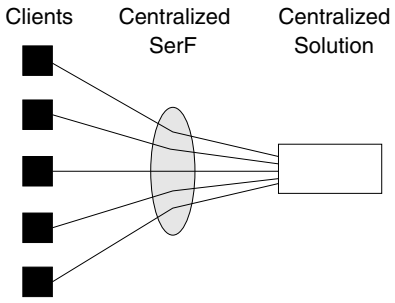


Figure 4. Distributed clients using a centralized solution

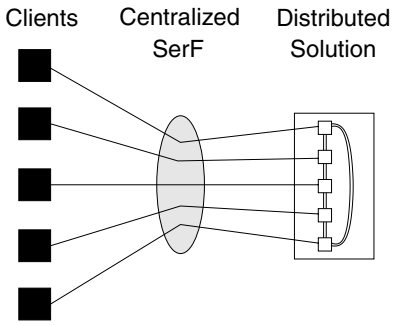


Figure 5. Distributed clients using a distributed solution but through a centralized gateway

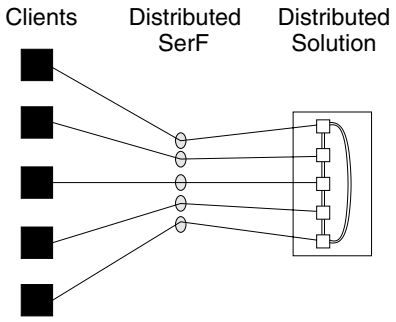


Figure 6. Distributed clients using a distributed solution

which (available) is the same for all proxies — **false**. This is so, because the resource is currently with the ResourceSerf. The ResourceSerf, thus transfers state by simply calling isRequested() on each of the current-generation proxy instances, and for each proxy for which requested is **true**, the ResourceSerf calls request on the corresponding next-generation proxy. The abstract state of the entire conflict resolution layer has been migrated to the next-generation protocol. Lines 6 & 7 in Segment 5 in setProtocolSerF are the lines that provide this behavior.

Instance Rebinding. As an interceptor between the client and its proxy in the mutual exclusion layer, the ResourceSerf has a chance to observe, and if necessary, block method invocations that the client makes on its proxy. This is achieved as follows. The client only holds a *logical* handle to the proxy. The actual reference is stored inside this handle, accessible only by the ResourceSerf. In addition, the ResourceSerf maintains a map from the logical handles to actual protocol proxy references. So there is a decoupling of the dependency between the logical handle that the client holds, and the actual protocol implementation. To complete the reconfiguration, after the above steps have been completed, ResourceSerf simply updates its map structure such that the client proxy handles now point to the new module instances instead of the old proxies. This is done in the last line of the loop body in Segment 5 in setProtocolSerF (Figure 3).

5 Dynamic Protocol Replacement

In this paper, we have extended the idea of dynamic module replacement to the domain of resource allocation in distributed systems. We have implemented two different protocols for solving mutual exclusion. The first one is a simple centralized queue-based protocol, where client requests are queued in a simple queue, and the client at the head of the queue gets access to the resource. In this case, we are dealing with a centralized solution to a distributed problem. All clients deal with a single ResourceSerf. This model is presented in Figure 4. This solution has its obvious drawbacks in performance penalties owing to no concurrency at all. The other protocol that we have implemented employs a distributed token ring [11] for resolving conflicts. This is a distributed solution that allows changes in topology at run-time. However, the module that controls client access to this protocol is still centralized (Figure 5).

Both the above approaches leave more to be desired. Even though the second case encapsulates a distributed protocol implementation, the fact that it is behind a centralized gateway makes it prone to problems — single point of failure, performance bottleneck, etc. The truly distributed solution, where a set of distributed clients are represented by a set of distributed proxies in the protocol, and moreover, the gateway itself is distributed is clearly the desired solution. Such a solution can tolerate faults, and does not involve a single bottleneck. A sketch of this solution is illustrated in Figure 6.

Serfs clearly have two separate functions — object creation, and object maintenance. In order for the distributed implementation to be possible, we separate these two functions. The main Serf still handles object creation; and rather

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE

than creating a logical handle that does not have any functionality on its own, creates a Serflet wrapper that encapsulates the actual product instance. The object maintenance function is now downloaded into the Serflet. In addition, the Serf manages the Serflets in a connected topology thereby enabling coordination among the Serflets, for instance, in the event of dynamic reconfiguration. This is the direction for future work.

6 Conclusion

In this paper, we have presented the use of the Service Facility design strategy as a language-neutral approach to module replacement in software. Furthermore, we have shown module replacement to be a powerful mode of dynamic reconfiguration, and how it can subsume other modes of reconfiguration as well. Finally, we have drawn all our contributions together through illustrative case examples demonstrating dynamic module replacement in a setting where it has not been well studied.

Dynamic reconfiguration has been widely researched, and yet, the results in academia have failed to cross over into widespread use by practitioners. Much of this delay in knowledge transfer can be attributed to a lack of support for dynamic reconfiguration in mainstream languages. In addition to decoupling other advanced object dependencies, this paper has illustrated how Serfs provide a material link to realizing module replacement in development environments.

References

[1] I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic adaptation and deployment of distributed components in HADAS. *TSE*, 27(9):769–787, 2001.

[2] T. Bloom and M. Day. Reconfiguration and module replacement in argus: Theory and practice. *IEE Software Engineering Journal*, 8(2):102–108, mar 1993.

[3] D. Duggan. Type-based hot swapping of running modules. In *International Conference on Functional Programming*, pages 62–73, 2001.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[5] M. P. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM TOPLAS*, 4(4):527–551, 1982.

[6] M. Hicks. *Dynamic Sotware Updating*. PhD thesis, University of Pennsylvania, Aug 2001.

[7] G. Hjálmtýsson and R. Gray. Dynamic C++ classes. In *Proc. of the USENIX 1998 Annual Tech. Conf.*, pages 65–76, Berkeley, USA, June 15–19 1998. USENIX.

[8] C. Hofmeister and J. M. Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Intl. Conf. on Distributed Computing Systems*, pages 101–110, 1993.

[9] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE TSE*, SE-11(4):424–436.

[10] L. Lamport and N. Lynch. Distributed computing: Models and methods. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 18, pages 1157–1199. Elsevier Science Publishers, 1990.

[11] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, California, 1996.

[12] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, 1989.

[13] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP 14*, pages 337–361, June 2000.

[14] Microsoft. *VS Live Conference*, San Francisco, 2002.

[15] M. L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. In *Proceedings of the 11th ICDCS*. IEEE CS.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[17] S. Robinson. *Advanced .NET Programming*. Number ISBN 1-861006-29-2. Wrox, 2002.

[18] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, Mar. 1993.

[19] N. Sridhar, S. M. Pike, and B. W. Weide. A pattern-oriented approach to dynamic module replacement: An experience report. Technical Report OSU-CISRC-7/02-TR17, CIS, OSU, Columbus OH, July 2002.

[20] N. Sridhar, B. W. Weide, and P. Bucci. Service facilities: Extending abstract factories to decouple advanced dependencies. In *Proceedings of ICSR-7*, pages 309–326, April 2002.

[21] SunMicrosystems. Java 2 on-line documentation. java.sun.com/products/jdk/1.2/ docs/api/index.html.

[22] B. Weide, S. Edwards, W. Heym, and T. Long. Characterizing observability and controllability of software components. In *ICSR-4*, pages 62–71, 1994.

Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)
1063-6927/03 $17.00 © 2003 IEEE