The following paper was originally published in the
Proceedings of the USENIX Annual Technical Conference (NO 98)
New Orleans, Louisiana, June 1998

Dynamic C++ Classes
A lightweight mechanism to update code in a running program

Gísli Hjálmtÿsson
AT&T Labs - Research
Robert Gray
Dartmouth College

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Dynamic C++ Classes

A lightweight mechanism to update code in a running program

Gísli Hjálmtýsson
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932
gisli@research.att.com

Robert Gray
Thayer School of Engineering[1]
Dartmouth College
Hanover, NH 03755
rgray@cs.dartmouth.edu

Abstract

Techniques for dynamically adding new code to a running program already exist in various operating systems, programming languages and runtime environments. Most of these systems have not found their way into common use, however, since they require programmer retraining and invalidate previous software investments. In addition, many of the systems are too high-level for performance-critical applications. This paper presents an implementation of *dynamic classes* for the C++ language. Dynamic classes allow run-time updates of an executing C++ program at the class level. Our implementation is a lightweight proxy class that exploits only common C++ features and can be compiled with most modern compilers. The proxy supports version updates of existing classes as well as the introduction of new classes. Our language choice and proxy implementation is targeted towards performance-critical applications such as low-level networking in which the use of C++ is already widespread.

1. Introduction

Every modern organization has software systems that are critical to its mission and must operate continuously. A network provider, for example, loses both revenue and customer goodwill if its switches or service controllers are temporarily unavailable. Because of the need for continuous operation, planned downtime is hard to schedule, and unplanned downtime can have cataclysmic effects. When a new service or security threat makes changes unavoidable, the changes are designed to minimize downtime, reducing system maintenance to repetitious patching. Such patching breaks the program's data abstractions and encapsulation, reduces modularity and increases coupling. At the same time, ignoring needed maintenance and development leads to an outdated system, one that eventually will become an obstacle to organizational development.

Complicating the situation is that, as network computing has transformed many industries, continuous change has become just as important as continuous operation. Rapid introduction of new functionality and dynamic adaptation to volatile needs is essential. Systems, particularly telecommunications systems, must be customized on a very fine time scale, either due to user demand or to load and usage fluctuations.

An effective way to allow both continuous operation and continuous change is through dynamic code updates. New code is added to a running program *without halting the program*, thus introducing new functionality but avoiding downtime. The idea of adding new code to a running program dates back to the earliest electronic computers, and dynamic linking [1,2,3,4,5,6,7,8] is now available for nearly all operating systems and programming languages. However, since languages such as C++ do not directly support the creation of dynamically loadable modules, preserving program-level abstractions across the dynamic-linking interface is difficult. In particular, current dynamic linkers break the type safety of C++,[2] since they are oriented towards functions rather than types and classes.

Recently, new environments and languages have been designed to dynamically download and execute programs [9,10]. In particular, Java [9,11] has become extremely popular and is in widespread use. However, although a Java *class loader* can lazily load the classes that make up an application, it has no knowledge of class versions and can only load each class once. In addition, the relatively poor performance of Java makes it impractical for low-level applications.

Here we propose *dynamic classes.* Dynamic classes allow new functionality to be introduced into an exe-

---

[1] This work was done at AT&T Labs – Research.

[2] We recognize that C++ is not "type secure" since the programmer can break any type abstraction through explicit casts. Current dynamic linkers, however, break type abstractions even if the programmer does *not* perform explicit casting.

cuting C++ [12] program without sacrificing type safety, performance or the object-oriented paradigm. Replacing an entire class, rather than individual functions, honors the semantic integrity of the program and minimizes interference between the update and the ongoing computation. Although comparable techniques exist in interpreted languages and agent-based environments, our objective is to use these mechanisms in telecommunication (and other) systems where performance is the primary concern. We outline a C++ proxy-based implementation that requires only existing features of the language and is usable with any complete C++ compiler.

Section 2 presents related work. Section 3 introduces dynamic classes. Section 4 describes our proxy-based implementation. Section 5 examines system performance. Section 6 presents several applications in which we are using dynamic classes. Sections 7 and 8 consider future work and our overall conclusions. Finally, the appendix contains psuedocode for our proxy class. The psuedocode is discussed in Section 3.

2. Related work

The idea of adding new code to a running program dates back to the earliest electronic computers, but perhaps the first structured approach can be found in the rudimentary dynamic linking of the Multics system [1]. Since then some form of dynamic linking has found its way into a wide range of programming languages, distributed-computing environments and even operating-system kernels. Since we are targeting the C++ language, we first consider various ways to add code to an executing C or C++ program, and then examine a few approaches that have been used in other languages and environments.

2.1 C and C++

Incremental linking, or runtime linking of code that was available to the compile-time linker, achieves lazy loading of the code and avoids resource allocation for code segments that are never used [13,14]. Conceptually, however, incremental linking is identical to traditional compile-time linking.

Dynamic linking, or runtime linking of code that was not available to the compile-time linker, truly supports the introduction of new functionality into a running program [2,3,4,5,6,7]. However, the C++ language does not directly support the creation of dynamically loadable modules, making it difficult to preserve program-level abstractions across the dynamic interface. In particular, current dynamic linkers break the type safety of C++, since they are oriented towards functions rather than classes. For example, to create an instance of a previously *unknown* derived class, the program must search a shared library for the desired constructor via a tedious C interface (e.g., `dlfind` on many systems), and then *cast* the resulting function pointer to the constructor type. In addition, some dynamic linkers do not allow a previously loaded code module to be replaced later unless every call into the module is made via the same tedious C interface. Even when the dynamic linker allows the *relinking* described in [15], there is no support for replacing a module that is in use or having multiple versions of a module coexist within the same program.

Dorward et al. [8] extend dynamic linking so that it preserves the type safety of C++ and works at the class level. Their solution allows a *new* derived class of a *known* base class to be dynamically loaded into a program. First, the shared library that contains the implementation of the new class is dynamically linked into the program. Then, a standard factory mechanism [16] is used to call into the library and create an instance of the new class (a preprocessor automatically generates the necessary factory routines). The instance is cast to the type of the base class and can be used wherever an instance of the base class is expected. Since all calls to the base class are type checked statically, and the base class constrains the derived class to have the same function signatures, type safety is preserved even though the program is actually invoking the operations of the derived class. Our implementation uses much the same mechanism to achieve type safety, but it extends Dorward's implementation by (1) allowing the replacement of a previously loaded class with a new version and (2) allowing multiple versions of a class to coexist within the same program. In other words, our implementation adds versioning.

The techniques of Hamilton and Radia [17] and Goldstein and Sloan [18] do allow multiple versions of a class to coexist within the same program. In the Hamilton and Radia approach, the program must be recompiled to take advantage of a new version (and hence stopped and restarted). Goldstein and Sloan, on the other hand, allow the new version to be dynamically added to the running program, but their solution is meant for distributed systems in which programs communicate by passing objects to each other. Since the libraries that the programs use might be upgraded at different times, a program might receive an object for which it does not contain the corresponding library version. When this happens, the program dynamically loads the appropriate library version and directs all accesses to the received object into this library. Their solution is intended to be completely transparent and does not provide application-level control over the active version (although it could be extended to provide

such control). In addition, their solution requires non-trivial compiler support and does not allow an instance of an old class version to be passed to code that was compiled against a newer class version.

2.2 Other languages and environments

Java is of particular interest due to its widespread popularity and availability. Java is an object-oriented language that is syntactically similar to C++ [9]. A Java program is made up of one or more classes. Rather than load each class into the Java virtual machine at program startup, a *class loader* dynamically loads each class at the time of first reference. Although the built-in class loader has no knowledge of class versions and will only load each class once, it would be possible to write a custom class loader that took versioning into account. This custom class loader might need help from (1) Java *interfaces* to cleanly separate interface and implementation, (2) a preprocessor to enforce a version-naming scheme, and (3) some proxy-like class. Java, however, simply does not provide sufficient performance for the low-level applications of interest. It will be worthwhile to reconsider Java once effective just-in-time compilation brings its performance closer to that of C++. If Java becomes an attractive choice for our target applications, the same dynamic-class mechanism presented in this paper could be reimplemented in Java.

The Limbo language, which is part of the Inferno system from Lucent technologies, is intended for the same type of distributed applications as Java [10]. A Limbo program consists of one or more modules. Each module consists of a public interface and a private implementation. The public interface can include functions, variables, data types and constants. These modules can be dynamically loaded, unloaded and reloaded at runtime. Although versioning support would need to be added at a higher level, this dynamic loading capability would play a large role in a dynamic-class implementation for Limbo. Like Java, however, Limbo is not in widespread use for the applications in question and does not yet provide the desired performance.

Many other languages - such as Eiffel, Lisp, Perl, Python [19], Scheme, SmallTalk [20], Standard ML and Tcl [21] - support some form of dynamic linking or loading. Depending on the language, the dynamic update can be as small as a single procedure, a single class or a single compilation unit. Unfortunately, many of these languages are interpreted and are too inefficient for performance-critical systems. None of them directly provide the necessary versioning support. Most importantly, none of them are in widespread use in our application environment.

Most agent-based environments – which include mobile-code systems [22,23], cooperative processes [24], and intelligent interfaces [25] - allow the dynamic introduction and removal of individual process entities or agents. There is little support, however, for replacing an existing agent while preserving ongoing agent conversations. In addition, many agent systems use interpreted languages that are not efficient enough for low-level processing. Finally, due to the communication overhead in current agent systems, an application is often implemented as a few large agents rather than many small agents. These large agents are too coarse of a replacement unit for many applications.[3]

Many distributed programming systems such as CORBA [26] and Argus [27] also allow the dynamic introduction and removal of individual processing entities. In CORBA this entity is a single process. In Argus this entity is a collection of objects and processes called a guardian. Both systems provide limited support for replacement. As in the agent case, however, a process or guardian is too coarse a replacement unit for many applications, particularly since a guardian is unavailable during the replacement process (i.e, all ongoing conversations are blocked). Bloom, however, does make the notable contribution of analyzing when it is safe to replace one implementation of a guardian with another [27], an analysis that could be applied directly to dynamic C++ classes.

Finally, Microsoft's component object model (COM) [28] allows the programmer to define components, which have an interface and a separate implementation. If the implementation changes, existing components will continue with the old implementation, while new components can use the new implementation. This approach is quite close to the approach that we use for our dynamic classes. In addition, COM is efficient enough that components can be used at the "class" level. On the other hand, COM has many additional features that we do not need; it has a longer learning-curve than our simple proxy; and it is not widely used in non-Windows applications. Like Java, however, COM will be worth revisiting as it evolves.

3. Dynamic classes

A *dynamic class* is a class whose implementation can be dynamically changed during program execution, allowing the introduction of new functionality at the class level. Of course, the main program must be able to communicate with (invoke the methods of) the new im-

[3] As we will see later, however, our dynamic classes can be used to *implement* agent replacement (in systems where agent replacement is sufficient).

plementation. Thus each implementation must have an interface that is known to the main program at compile time. Each new implementation either updates an existing class (a new version) or introduces a new class (a new type) that uses the known interface.

3.1 Version Update Semantics

A basic problem when updating an existing dynamic class is what to do with existing objects. There are at least three approaches as shown in Figure 1. One approach (a) is to raise a "barrier," blocking object creation until all existing objects of older versions have expired. Then the new version takes over and object creation resumes. This approach is conceptually equivalent to halting, modifying and restarting the system. Another approach (b) is to *recreate* all existing objects using the new version. This approach retains a crucial property of raising a barrier, namely that at any time all objects of a particular class are of the same version. On the other hand, since different class versions can have different internal data structures, copying each object's state requires an understanding of the object's *semantics*.

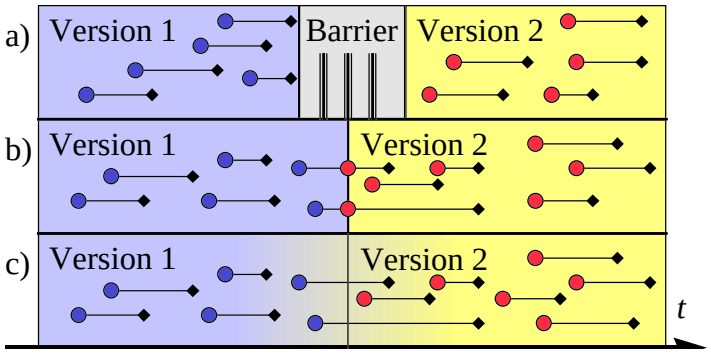

Figure 1: Three approaches to version updating

The last approach (c) is not to take any action at all. All new objects are created with the new version, and existing objects continue with their current versions. Once the existing objects finish their tasks and are destroyed, only the new version will be in use. We adopt this last solution since it is the most basic, is sufficient in all the cases that we have considered, and can be implemented efficiently. However, we also include a method that an object can use to determine if its version is the most recent version, allowing the programmer to explicitly migrate objects of a particular class. The programmer would need to write state-capture and restoration routines for each class version. These routines would produce and accept version-independent representations of an object's state.

3.2 The Interface Semantics

To allow "hot" updates, an interface monitor screens every message that passes through the dynamic class interfaces. This monitor is conceptually a class proxy. For each dynamic class, the monitor maintains a map that associates the class name with the current implementation version (and its location on external storage). A dynamic class is *invalid* if the running program does not contain any implementation version for that class. All dynamic classes start out as invalid. When a message is sent to an invalid class, the monitor locates and loads the current implementation version, updating the internal map as needed. The monitor then passes the message on to the newly loaded version.

The interface monitor provides three methods to manipulate the current version: *activate*, *invalidate* and *activate-and-invalidate*. The *activate* method registers a new version as the current version. Objects of older versions remain in existence. Once all old objects have expired normally, the older versions are removed from the system. The *activate-and-invalidate* method registers a new version as the current version but also invalidates (destroys) all objects of older versions. C*onversations*[4] in which these objects were engaged are broken. The surviving participants must recover and retry. The *invalidate* method invalidates (destroys) all objects of a given version. Invalidating the current version is an error. With these semantics, the three methods maintain the invariant that each dynamic class has a unique current version.

4. Implementation

We had three design goals for our C++ implementation of dynamic classes: (1) efficiently find and invoke the correct version of each method, (2) hide the dynamic class mechanism as much as possible from the end programmer, and (3) use only standard C++ features. The implementation should not require special preprocessor or compiler support, nor depend on *compiler-specific* details. These design goals led us to proxy classes. In this section, we first present our proxy-class implementation, and then discuss the tradeoffs involved.

---

[4] We use the term *conversation* for a sequence of message exchanges (or method invocations) between two objects. In our implementation, if one of the objects is dynamic and is invalidated, the other object will receive an exception when it attempts to invoke the invalidated object's methods. This other object must then perform some application-specific recovery, such as constructing a new object of the same class as the invalidated one. Clearly the programmer should think carefully before using invalidation.

The core of our implementation is a generic template class. The template, which is shown in the appendix, uses only standard C++ features and can be compiled with any complete C++ compiler. The template serves as a proxy (or *smart pointer*) for each dynamic class. As with any proxy, a program creates a dynamic class instance by creating a proxy instance instead.

Each dynamic class is written as two separate parts: (1) an abstract interface class that is known to the program at compile time and (2) one or more implementation classes that inherit from the interface class. There is one implementation class for each version of the dynamic class. The abstract interface class specifies the public operations that remain constant across all versions of the dynamic class. These operations are defined as pure virtual functions so that each derived implementation class is forced to provide them. In addition, although each implementation class can have any additional methods and data members that it needs to perform its task, only the operations defined in the interface class can be called from other program modules. This only makes sense since otherwise a program module might become dependent on a particular version of the dynamic class. Each implementation class is compiled into a separate shared library.

To use a dynamic class, the template is instantiated on the interface class. At run-time, the template locates the shared library that contains the most recent implementation class and loads this library into the program's address space. The template calls into the library to create an instance of the implementation class, and casts the instance to the type of the interface class. Finally, the public interface operations are accessed through the template using standard pointer redirection.

The template also provides static methods that implement *active*, *invalidate* and *activate-and-invalidate*. Most software systems will provide an external interface through which an administrator, developer or automated management tool can invoke these methods.

As an example, consider a dynamic class whose job is to receive packets sent across a network connection. For simplicity, the dynamic class interface provides only a single operation.

```
class Receiver {
  public:
    virtual Packet receivePacket (void) = 0;
}
```

The programmer might write two implementation classes, the normal production version and later, after the discovery of an unexpected problem, a debugging version that contains new debugging code.

```
class ReceiverImp: public Receiver {
  public:
    Packet receivePacket (void) {…}
}

class ReceiverDebuggingImp: public Receiver {
  public:
    Packet receivePacket (void) {
      logDebuggingInfo(); …
    }
}
```

Each of these two implementation classes is compiled into its own shared library, say *imp.so* and *debugimp.so* respectively. Using the dynamic class is now straightforward (*dynamic* is the name of our proxy template as shown in the appendix).

```
// normal program operation – create and use
// normal packet receivers
dynamic<Receiver>::activate ("imp.so");
dynamic<Receiver> receiver;
Packet packet = receiver.receivePacket();
…
// switch to debugging mode in response to
// some external event (library name would
// be included in the external event)
dynamic<Receiver>::activate ("debugimp.so");
// now all new packet receivers will contain
// the debugging code
dynamic<Receiver> otherReceiver;
…
```

Thus new debugging functionality is introduced without stopping the running program. In addition, once the bug is identified, the developer can create a third version of Receiver, one that contains the necessary fix. The fixed version can then be activated, again without stopping the running programming. Of course, we do not intend to suggest that all bugs can be fixed without stopping the program, but at least some bugs can be. For example, our Receiver might simply be mistranslating a particular kind of packet.

4.1 Implementation details

Construction and invocation. Since there can be multiple versions of a dynamic class within a program, there can be multiple implementations of each method within the program's address space. The correct implementation must be called when a method is invoked on a particular object. The problem is to have a method invocation that is version-dependent, but that can be resolved at compile time by a C++ compiler that (1) is oblivious to versioning and (2) has access to only to the interface class. The solution is a two-level indirect method resolution at runtime. The first level is the version-dependent mapping, which we implement ourselves inside the dynamic class proxy; the second level is the method mapping within a version, which we can achieve with standard C++ virtual methods (and the associated vtables).

In our approach, the version of each object remains unchanged for the lifetime of the object. Therefore, the version mapping can be resolved during object creation

and stored in the instance of the proxy. In fact, once the object is created, the needed mapping is just a single pointer to the new object (namely the `object` pointer that appears in the template definition in the appendix).

The method mapping is achieved in C++ by defining all methods in the interface class as virtual. C++ adds a vtable to each derived implementation class, and resolves all method calls through the vtables [12]. Figure 2 illustrates the two-level mapping. The proxy contains a pointer to the object; the object contains a pointer to the vtable; and the vtable contains pointers to the methods of the object's implementation version.

Since the vtable is not used when invoking a constructor, the problem of how to actually construct the object remains. Since each implementation class has (1) its own constructors and (2) possibly a different size, normal C++ constructor syntax can *not* be used. Instead we use the standard factory pattern and require each class version to provide a static method, *createInstance*, which the proxy calls instead of the constructor. A side effect of this approach is that each dynamic class essentially has only one constructor, namely the one that *createInstance* chooses to invoke. Additional initialization must be done through other methods.

External map. Each version of a dynamic class is compiled into its own shared library. At runtime, the *activate* and *activate_and_invalidate* methods must be able to locate the correct library given some symbolic name for the desired class version. There are several ways to accomplish such a mapping, but we found that the easiest approach for the end programmer was to simply use the library name (without its path) *as the symbolic name*. The two methods then search all known library directories for the given library.[5] Of course, the library name does not need to be known to the program at compile time; it can be passed to the program at runtime during the version-update process.

We have also extended our proxy so that the symbolic name can be an arbitrary URL, both to support network applications and to use Web technology for the storage of dynamic class libraries. If the URL refers to a library on the local machine, the library is immediately loaded into the program's address space. If the URL refers to a library on a remote machine, the library is first downloaded onto the local machine. A further description of our network support is beyond the scope of this paper. Relevant issues include caching the libraries on the local machine, handling different machine architectures,

[5] How library directories are specified is platform dependent. On most Unix machines, library directories are listed in the environment variable LD_LIBRARY_PATH.

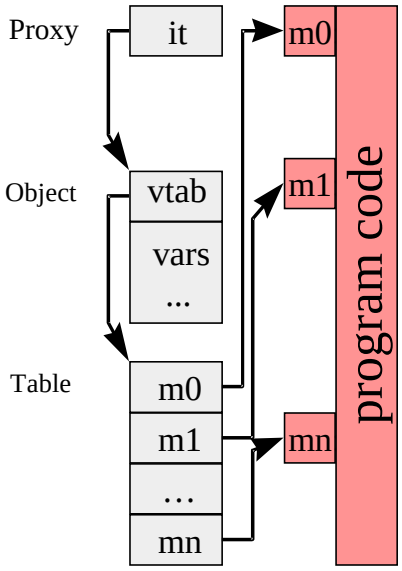

Figure 2: Method invocation

checking the security credentials of the downloader, and imposing a consistent, meaningful naming scheme.

Template versions. The dynamic class system provides three versions of the proxy template. One version has restricted functionality but high performance; another has full functionality but slightly lower performance; and the third falls between the first two on both functionality and performance. The full-functionality version allows different dynamic classes to share the same interface class, and therefore allows the introduction of *new* dynamic classes into the system. For example, a programmer could write a *NetworkConnection* interface class, and then write the dynamic classes *TcpipConnection*, *RpcConnection*, etc., which all use *NetworkConnection* as their interface. *TcpipConnection*, *RpcConnection*, etc., could all have multiple versions. Thus this version of the template provides two "mappings''. For each dynamic class, there is a list of the class versions that are currently present inside the program, and for each interface class, there is a table that associates each dynamic class name with the correct version list. This full-functionality version of the template is shown in the appendix. Note that *activate*, *invalidate* and *activate_and_invalidate* take two parameters, the name of the dynamic class and the name of the library that contains the desired version of that class. Note also that *activate* and *activate_and_invalidate* return a unique handle for the dynamic class, which is passed to the template constructor to identify the desired dynamic class. Handles make the constructor much faster, since looking up a handle is much more efficient than looking up a class name.

The medium-functionality template requires each dy-

namic class to have its own interface class (and thus supports version updates only). The template still maintains a version list for each dynamic class, but eliminates the class-name to version-list mapping, avoiding (1) the string lookup in the *activate*, *invalidate*, and *activate_and_invalidate* methods and (2) the handle lookup in the constructor. In this template, the three methods take only the library name as a parameter, and the constructor takes no arguments. The class name, which is needed to invoke the constructor, is obtained from a special function in the shared library (i.e., a static function that returns the class name as a `const char *`). The medium-functionality template is the most commonly used template. Its performance is discussed in Section 5.

Finally, the low-functionality template does not support the *invalidate* and *activate_and_invalidate* methods. New versions can be introduced, but old versions can not be invalidated. Since each version remains valid as long as objects of that version exist, the redirection operator does not need to check for invalid versions. More importantly, the redirection operator no longer throws exceptions, which allows C++ compilers to inline the operator and eliminate one function call. Together these two things make method invocation much faster as will be seen in Section 5. All three versions of the template can be freely used in the same program.

4.2 Implementation tradeoffs

We chose the proxy approach since the ability to use standard C++ development environments was a primary design goal. If we had been willing to sacrifice this goal, we would have had more implementation choices, most notably a custom preprocessor. A sufficiently complex preprocessor could (1) make dynamic classes look more like normal C++ classes (i.e., multiple constructors per dynamic class and method access via selection (.) rather than just deference (`->`)), and (2) provide better performance. A careful analysis of the tradeoffs, however, indicates that the preprocessor advantages are not as great as they first appear, and do not justify the additional complexity and development time.

Performance. The proxy approach involves two overheads. First, the proxy constructor must find and invoke the correct factory method. The factory method then invokes the real constructor. Second, all method accesses go through the proxy's dereference operator, which involves an extra function call (in the absence of inlining) and a Boolean comparison to verify that the class version has not been invalidated. Although custom preprocessor support could not eliminate this entire overhead (i.e., there must be a mapping from an object to its version), it would eliminate at least some levels of indirection. Here, however, it is worth considering that a typical dynamic class will not be the finest-grained class in the system. It is unlikely that a programmer will need a dynamic *Point* class if a *Point* is simply a coordinate in two-dimensional space. It is more likely that the programmer will need a dynamic *Renderer* class that draws some figure given a set of *Points*. The methods in *Renderer* would do significant processing, and the extra time needed to just invoke the methods would be insignificant. Our expectation is that most (if not all) dynamic classes will fall into the same category as *Renderer* and perform nontrivial processing in most of their methods. All of the application work so far confirms this view. For this reason, we feel that the performance issues are minor, and do not justify any custom preprocessor or compiler support. If our expectations are wrong, however, and programmers start to make featherweight classes dynamic, we will need to re-examine our proxy approach. We will say more about performance in Section 5.

One technique that does not involve any custom support is to re-map the vtable associated with a particular dynamic class version in response to certain events. For example, the *invalidate* operation could make every entry in the vtable point to a dummy method that throws an exception. Then the dereference operator would not need to check if the class version were still valid. The proxy is still necessary, however, since the proxy provides additional functionality behind the scenes. For example, the proxy maintains a count of all objects of a particular version, so that the version code can be removed from the program's address space as soon as those objects have been destroyed.[6] Thus, given that (1) the proxy is still necessary, (2) working with vtables does introduce a few compiler dependencies, and (3) the performance penalties without re-mapping are small, we believe that vtable re-mapping will not provide sufficient benefits to be worthwhile.

Abstraction hiding. The proxy approach means that the program (and hence the programmer) must explicitly know about dynamic classes at some level. In addition, the proxy approach prevents the usage of some standard C++ constructs. Most notably, a dynamic class can have only a single constructor, and all method access must take place through the dereference operator.[7] Finally, our proxy implementation allows a programmer to obtain a direct reference to the embedded object (through a perhaps atypical use of the dereference operator). The programmer could then access the

[6] Removal occurs only if the version is no longer the active version.

[7] Passing a *reference* to a dynamic class is fine, but method access still involves the dereference operator.

object without going through the proxy class. Preprocessor and compiler support could address all of these problems. However, it is a simple C++ programming task to write a wrapper class that contains an embedded instance of our dynamic-class proxy. This wrapper class could (1) hide the existence of the proxy from the rest of the program, (2) provide multiple constructors, (3) support normal C++ access syntax, and (4) prevent the client code from obtaining any direct reference to the actual versioned object. For example, here is a wrapper for our Receiver class (assuming that the interface has been extended with an *initialize* method).

```
class ReceiverWrapper {
  private:
    dynamic<Receiver> *receiver;
  public:
    ReceiverWrapper (void) {
      receiver = new dynamic<Receiver>;
    }
    ReceiveWrapper (int packetType) {
      receiver = new dynamic<Receiver>;
      receiver -> initialize (packetType);
    }
    Packet receivePacket (void) {
      return (receiver -> receivePacket());
    }
}
```

The wrapper class looks like a normal class to the rest of the program. The only exception is the program module that accepts versioning instructions from external sources (and passes these instructions on to the appropriate proxies). Given the ease with which these wrapper classes can be written, we again felt that custom preprocessor or compiler support was not justified. Of course, the fact that a wrapper class can be written does not mean that it will be written. The programmer is free to bypass our proxy methods, breaking the dynamic-class abstraction. It is hard to imagine how the programmer could do this accidentally, however, and no programmer has done it accidentally so far. Thus we are content to provide a flexible mechanism without *enforcing* all usage requirements.

In a similar vein, existing programs cannot use our dynamic classes without modification. In a large software system, however, it is likely that these modifications would be confined to a particular subsystem, already hidden behind an appropriate class. In addition, it is difficult to imagine that an existing program could use *any* dynamic class implementation without modification. Unless we have a preprocessor or compiler that essentially makes all classes dynamic, we must at least add some syntactic markup and then recompile.

Finally, the behavior of static methods in a dynamic-class interface is currently undefined. This is mainly an implementation detail. The static methods must be compiled only into the main program, not into any of the shared libraries. Then all static method invocations will be directed to the same code (which is what we want since the methods will only make sense if they perform version-independent processing). Compiling the static methods only into the main program does not require any special support, although some preprocessor support would help the programmer avoid mistakes.

Inheritance. The proxy approach complicates inheritance in several ways. First, the proxy approach demands a clean separation between the interface and implementation of a class, simply because the interface and implementation *must be* separate classes. Unfortunately, such a clean separation is not seen in many existing C++ programs. On the other hand, any dynamic class implementation will require the same separation, since a dynamic class must have a version-independent interface. Otherwise client code would quickly become dependent on particular versions.

Second, the proxy restricts how dynamic classes are inherited. In general, interface classes inherit from other interface classes; an implementation class inherits from other implementation classes; and a *normal* class that wants to extend the functionality of a given dynamic class *contains* an instance of the appropriate proxy. Preprocessor or compiler support could certainly relax this strict separation. In the same light as some of the other issues, however, it is questionable whether such a relaxation should be allowed. Imagine, for example, if a dynamic class inherits from a particular version of some other dynamic class. Then, whenever the system constructs an instance of the subclass, it must identify and use the correct version of the superclass (and of the superclass of the superclass). Although there are programmers who could keep the resulting dependencies straight, the same inheritance effect can be achieved much more cleanly and easily with separate interface and implementation hierarchies (and without complex compiler or preprocessor support).

Third, our proxy does have undesirable consequences for interface polymorphism. Even if one *interface* inherits from another, the *proxy class* instantiated on the derived interface is *not* related to the *proxy class* instantiated on the base interface. Therefore, contrary to the intent of the interface inheritance, the proxy of the derived interface cannot be used where a proxy of the base interface is expected. Our solution is to provide a template function that performs an explicit cast; the template function succeeds (at compile time) only if the respective interfaces are related through inheritance. Although our solution is sufficient, simple preprocessor support would be useful here.

Finally, the full implementation of any superclasses must be compiled into the same library as the dynamic

class version to ensure that all superclass references are resolved correctly (at compile time). Although this means that the superclass code might appear in multiple libraries, it simplifies our implementation significantly and involves minimal extra work for the programmer. Again some preprocessor support would be useful here.

Summary. In contrast with a preprocessor- or compiler-based approach, our proxy solution is much simpler, but has lower performance, does not fully hide the dynamic class abstraction, and restricts inheritance. However, the performance penalty is almost always small relative to the processing that the dynamic classes are performing; the proxy can be hidden completely with a straightforward wrapper class; and *any* dynamic-class implementation will likely restrict inheritance so that inheritance remains understandable.

5. Performance Evaluation

5.1 Time Complexity

We ran two tests, one measuring the time to construct and destroy an object (the class had no data elements and its constructor had no arguments), and the other measuring the time to invoke an object method (the method had no arguments and no return value). Each test involved three cases: (1) a standard stack-allocated class that does not have virtual methods, (2) a standard heap-allocated C++ class that has virtual methods and (3) a dynamic class created through our medium-functionality template. We considered both case (1) and (2) since a dynamic class always involves virtual functions and heap allocation. Each test was compiled with the SGI C++ compiler and was performed ten million times per run for ten runs on an SGI Indy.

When the constructor is empty, the construction overhead of dynamic classes is 80% versus the heap-allocated class, and 1091% versus the stack-allocated class. When the constructor zeroes out a 128-byte block of static memory, the overheads drop to 14% and 160% respectively. In a multi-threaded environment, the time to acquire a lock (to prevent corruption of the version lists) dominates the construction process, and the overheads increase to 650%/119% (empty/non-empty constructor) versus the heap-allocated class, and 4872%/160% versus the stack-allocated class.

When the method is empty, the invocation overhead of dynamic classes is 240% versus the heap-allocated class, and 611% versus the stack-allocated class. When the method defines three local variables, increments the value of these variables by one, and performs three integer comparisons (that evaluate to false), the overheads drop to 110% and 175% respectively. In addition, if *invalidation* is not required, the low-functionality template can be used instead of the medium-functionality template. The low-functionality template has much better performance, since its redirection operator does not throw an exception and can be inlined by the compiler.[8] With this template, the overheads are 39%/18% (empty/non-empty method) versus the heap-allocated class, and 191%/55% versus the stack-allocated class.

The performance penalty of making an existing C++ class dynamic is high if its methods are (nearly) empty. The penalty is quite low, however, if its methods do any nontrivial processing. Thus our dynamic-class implementation is not appropriate for a low-level class such as *Point*, since *Point* is (1) computationally trivial (each method does little more than a single assignment) and (2) small enough to allocate on the stack. In addition, if *Points* are used throughout the system, they will be created and destroyed constantly, leading to a severe performance penalty since the proxy constructor takes much longer than the simple *Point* constructor. However, our dynamic-class implementation is appropriate for most higher-level classes such as the *Renderer*, especially if the class objects (1) are allocated on the heap anyway, (2) are created and destroyed infrequently, or (3) do nontrivial processing in their constructors and other methods. As discussed above, we expect that dynamic classes will be used only with high-level classes anyway. Our implementation provides excellent performance for these classes. In all of the applications that we have implemented at AT&T, for example, the dynamic-class methods do far more processing than the test cases presented in this section, and the overheads are insignificant.

5.2 Space Complexity

The space requirements of dynamic classes are low. For each *class*, the proxy maintains a version list, a pointer to the active version (so that it does not have to search the list during construction), and a synchronization lock (in a multi-threaded environment only). The version list has one entry per version. Each entry contains the associated class and library names, a flag that indicates whether the version is the active version, and a count of the number of objects of the version. For each *object*, the proxy maintains two pointers, one to the actual object and one to the object's version information inside the version list. In addition, since a dynamic class always has virtual functions, the actual implementation object has a pointer to the appropriate vtable.

Thus the *per-object* space overhead of dynamic classes is three pointers (including the vtable pointer), and dy-

[8] Most C++ compilers will not inline a method that can throw an exception.

namic classes can be used only if this overhead is acceptable. For the *Point* class, the overhead is probably unacceptable, since the three pointers might take up more space than the *Point*'s actual data. For the *Renderer* class, the overhead is probably acceptable. In the applications that we discuss below, the three pointers are less than 10% of the total object size.

6. Applications

Our main motivation for this work is network-control and service-management applications that (1) demand the high performance of C++ but (2) must operate continuously. We have used dynamic classes in three such applications at AT&T, namely mobile agents, control-on-demand and connection management. The *mobile-agent* application is a building block for the control-on-demand application, but also demonstrates the viability of native code in a heterogeneous environment. The *control-on-demand* application uses agents to inject application-specific control policies into a router. Finally, the *connection-management* application uses dynamic classes to inject handlers for new connection types into a running connection manager.

6.1 Mobile Agents

A mobile agent is an executing program that can migrate *at times of its own choosing* from machine to machine in a heterogeneous network [22]. Mobile agents are attracting growing attention as a means to easily realize complex, distributed applications, and are being used at AT&T in several prototype network-management applications. We have implemented an efficient mobile-agent system on top of our dynamic-class mechanism. Each agent is a version of the same dynamic class. The interface class defines the operations that are common to all agents, most notably *migrate*, *captureState*, *restoreState* and *run*. Unlike most dynamic classes, the interface class *implements* the *migrate* operation. The agent implements the other three. Like all dynamic classes, each agent is compiled into its own shared library.

A mobile agent starts executing when a bootstrap program loads the agent (via the dynmic-class mechanism) and calls its *run* method. The *run* method performs the agent's task. If the *run* method decides that the agent should migrate to another machine, it calls the *migrate* method. The *migrate* method calls the *captureState* method to package up the agent's current state, and then transmits the state image and the *URL* of the agent's shared library to a server on the target machine. The server downloads the shared library, loads the agent (again via the dynamic-class mechanism), calls the *restoreState* method to restore the agent's state, and finally calls the *run* method. The *run* method continues with the agent's task, checking the agent's current state to decide what to do next.[9]

6.2 Control on demand

Control-on-demand [29] is flow-oriented active networking. Applications inject customized control policies for each flow into the network routers. These policies exploit strategic positioning, local network knowledge and application semantics to improve the performance or perceived quality of the flow. They may act both in the control plane and the data plane. The former supports connectivity control, such as floor management for a teleconference or advanced group management for a multicast. Applications of the latter include (1) stream thinning at a branch point in a variegated multicast, (2) discarding less important packets during congestion (e.g., discarding B and P frames to protect I frames in an MPEG stream), or (3) monitoring packet loss and retransmitting lost packets from inside the network.

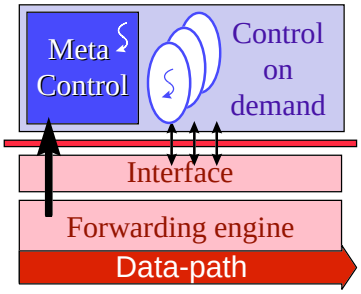

Figure 3: A node supporting control-on-demand

Figure 3 shows a control-on-demand network node. The two major parts are a forwarding engine (below) and controllers (above) separated by an opaque interface. The meta-controller accepts, installs and runs new controllers on demand, using dynamic classes in the same way that they were used for mobile agents.

6.3 Service/connection management

To experiment with dynamic classes in service-management applications, we have built a prototype connection manager. Before transmitting, an application issues a connection request to the connection manager. The manager creates a connection instance, performing admission control and other "book-keeping" in the process. On disconnect, the instance is destroyed, and resources allocated to the connection released. While the connection is active, the instance is responsible for ensuring service quality.

---

[9] Since different machine architectures cannot use the same shared library, the agent must be pre-compiled for all machine architectures on which it might find itself. Each machine inserts its architecture "name" into the URL before downloading the library. This naming scheme, as well as other issues such as security and fault-tolerance, is beyond the scope of this paper.

The static part of the connection manager (the main program) recognizes only a general interface to connection objects. Type-specific implementations of this interface are introduced dynamically (using the full-functionality version of dynamic classes). Figure 4 depicts a hierarchy of connection types that is dynamically constructed during the execution of the connection manager. The manager is an event processor that handles events of the form: {*connection-type name, data (initial state), code reference (URL)*}. If the connection type is not known, the code is introduced as a new type. If the type is known, but the code reference has changed, the code is introduced as a new implementation version. In either case, the new code is retrieved and installed using the dynamic-class mechanism.

7. Future work

The first area of future work is to add security mechanisms to our dynamic-class implementation so that a program can verify the origin of the dynamic classes that it is instructed to load.

A second area of future work is to handle the case where several dynamic classes must undergo a version update as an atomic unit. Potential solutions include a simple transaction mechanism or a constraint definition language (i.e., new versions can be loaded at any time, but become active only when all constraints are met).

Finally, we are working with other groups at AT&T to identify additional applications for dynamic classes. We also plan to provide a dynamic-class implementation for Java. Although Java is unsuited for low-level control software, it can be used to implement higher-level components in telecommunications systems.

8. Conclusion

Dynamic classes provide powerful support for the maintenance and extension of mission-critical and other long-running applications. New implementations of a class can be dynamically added to and removed from a running program, eliminating the need to bring down the program when fixing bugs, enhancing performance, or extending functionality. The implementation discussed in this paper provides an easy-to-use dynamic-class library for the C++ language. The implementation preserves type safety and the class abstraction. It does not require special compiler support, and is efficient enough for use in low-level software.

We have already used dynamic classes in a mobile-agent application and as part of a larger programmable-network effort. In our experience, dynamic classes are efficient, easy to use and sufficient for most tasks.

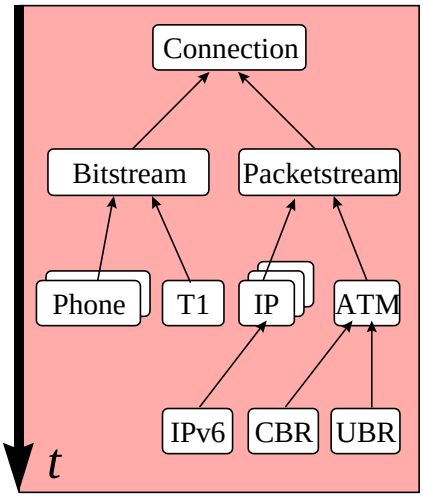

Figure 4: A hierarchy of connection types

9. Availability

Our dynamic-class implementation is available for Irix, Solaris and Windows 95/NT. It is known to compile under Irix with release 4.0 of the AT&T C++ compiler, under Solaris with release 4.2 of the Sun C++ compiler, and under Windows 95/NT with release 4.2 of the Microsoft C++ compiler. Porting to another platform is simple, as long as the platform has (1) a C++ compiler with exception and template support and (2) dynamic-linking facilities comparable to those of Irix. Interested readers should contact the second author.

10. Acknowledgments

Many thanks to the anonymous reviewers and our shepherd, Benjamin Zorn, for their excellent feedback, and to the AT&T programmers who have put our dynamic-class implementation through its paces.

Appendix

The following C++ code is a simplified version of the full-functionality template. As discussed in the paper, this template allows different dynamic classes to share the same interface class. Other templates support version updates only to achieve higher performance (by eliminating a table lookup). The template methods all throw C++ exceptions on error.

```
typedef list<DynamicVersion*> VersionList;
typedef int Handle;

template<class T> class dynamic {
    // the actual object, its version
    // information and its class handle
    T *object;
    DynamicVersion *dvPtr;
    Handle classHandle;
  public:
    // constructors, etc.
```

```
    dynamic(Handle handle);
    dynamic(const dynamic<T>& proxy);
    ~dynamic();
    dynamic<T>& operator= (
      const dynamic<T>& proxy);
    // smart pointer
    T *operator-> (void);
    // activate a version
    static Handle activate (
      const char *libraryName,
      const char *className = NULL);
    // invalidate a version
    static void invalidate (
      const char *libraryName, Handle handle);
    // activate and invalidate
    static Handle activate_and_invalidate (
      const char *libraryName,
      const char *className = NULL);
  private:
    // static data (shared by all versions and
    // classes implementing the interface <T>)
    // 1. map: handles to version list
    static VersionList **versionMap;
    // 2. map: handles to active version
    static DynamicVersion **activeMap;
    // … more …
};
```

References

[1] F. J. Corbato and V. A. Vyssotsky, "Introduction and Overview of the Multics System," Proceedings of the AFIPS Fall Join Computer Conference, 1965, pp. 185-196.

[2] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. "Shared Libraries in SunOS.," *Proceedings of the USENIX Summer Conference*, 1987, pp. 375-390.

[3] James Kempf and Peter B. Kessler, "Cross-Address Space Dynamic Linking," Technical Report TR-92-2, Sun Microsystems Laboratories, Inc., Mountain View, California, 1992.

[4] W. Wilson Ho and Ronald A Olsson. "An approach to genuine dynamic linking," Software-Practice And Experience, volume 21, number 4, April, 1991, pp. 375-390.

[5] Donn Seeley. Shared Libraries as Objects. USENIX Summer Conference Proceedings, 1990, pp. 25-37.

[6] March Sabaella. "Issues in Shared Library Design," USENIX Summer Conference Proceedings, 1990, pp. 11-23.

[7] Michael Franz. "Dynamic linking of software components," IEEE Computer, volume 30, number 3, March, 1997, pp. 74-81.

[8] Sean M. Dorward, Ravi Sethi and Jonathan E. Shopiro. "Adding New Code to a Running C++ Program," Proceedings of the USENIX C++ Conference, 1990, pages 279-292.

[9] "The Java Language: A White Paper," Sun Microsystems White Paper, Sun Microsystems, 1994.

[10] "Inferno: la Commedia Interattiva," Lucent Technologies White Paper, Lucent Technologies, Inc., 1997.

[11] Mary Campione and Kathy Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet*, Addison-Wesley, 1996.

[12] Bjarne Stroustrup. The C++ Programming Language (3rd Edition), Addison Wesley, 1997.

[13] J. J. Puttress and H. H. Goguen, "Incremental Loading of Subroutines at Runtime," Technical Report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1986.

[14] R. W. Quong, "The Design and Implementation of an Incremental Linker," Technical Report CSL-TR-88-381, Computer Systems Laboratory, Stanford University, 1989.

[15] David Keppel and Stephen Russell. "Faster Dynamic Linking for SPARC V8 and System V.4," Technical Report 93-12-08, University of Washington, 1993.

[16] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[17] G. Hamilton and S. Radia. "Using Interface Inheritance to Address Problems in System Software Evolution," *Proceedings of the ACM Workshop on Interface Definition Languages*, 1994.

[18] Theodore C. Goldstein and Alan D. Sloane. "The Object Binary Interface – C++ Objects for Evolvable Shared Class Libraries," Technical Report TR-94-26, Sun Microsystems Laboratories, Inc., Mountain View, California, 1994.

[19] Mark Lutz. *Programming Python*. O'Reilly, 1996.

[20] A Goldberg. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, Massachusetts.

[21] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

[22] James E. White. "Telescript Technology: The Foundation for the Electronic Marketplace," General Magic White Paper, General Magic, Inc., 1994.

[23] Robert S. Gray. "Agent Tcl: A Flexible and Secure Mobile-Agent System," Mark Diekhans and Mark Roseman, editors, *Proceedings of the 4th Annual Tcl/Tk Workshop*, Monterey, California, July, 1996.

[24] Michael R. Genesereth and Steven P. Ketchpel. "Software Agents," Communications of the ACM, 37(7), July, 1994, pages 49-53.

[25] Yezdi Lashkari and Max Metral and Pattie Maes. "Collaborative Interface Agents," *Proceedings of AAAI '94*, 1994

[26] Jon Siegel, *CORBA: Fundamentals and Programming*, Wiley, 1996. ISBN 0471-12148-7.

[27] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System* Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983.

[28] "The component object model (COM): A technical overview," Microsoft White Paper, Microsoft, Inc., 1996.

[29] Gísli Hjálmtýsson and Samrat Bhattacharjee. "Control on Demand - Customizing Control for Each Application," AT&T Technical Memorandum, 1997.