An Approach to Implementing Dynamic Adaptation in C++

Scott D. Fleming, Betty H.C. Cheng, R. E. Kurt Stirewalt and Philip K. McKinley

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{sdf,chengb,stire,mckinley}@cse.msu.edu.

ABSTRACT
This paper describes TRAP/C++, a software tool that enables new adaptable behavior to be added to existing C++ programs in a transparent fashion. In previous investigations, we used an aspect-oriented approach to manually define aspects for adaptation infrastructure, which were woven into the original application code at compile time. In follow-on work, we developed TRAP, a transparent weaving technique for automatically generating adaptation aspects, where TRAP/J is a specific instantiation of TRAP. This paper presents our work into building TRAP/C++, which was intended to be a port of TRAP/J into C++. Designing TRAP/C++ required us to overcome two major hurdles: lack of reflection in C++ and the incompatibility between the management of objects in C++ and the aspect weaving technique used in TRAP/J. We used generative programming methods to produce two tools, TrapGen and TrapCC, that work together to produce the desired TRAP/C++ functionality. Details of the TRAP/C++ architecture and operation are presented, which we illustrate with a description of a case study that adds dynamic auditing capabilities to an existing distributed C++ application.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Code generation
General Terms: Code generation
Keywords: Dynamic adaptation, middleware, program families

1. INTRODUCTION
A dynamically adaptive program is a program that contains facilities for selecting and incorporating new behaviors at run time. Dynamic adaptation is increasingly important, as the computing and communication infrastructure continues to expand and diversify. With the rise of the “Mobile Internet”, software on compact, wireless networked, constantly moving computers such as handhelds and two-way radios must change their behavior in response to dynamic conditions and changing policies, while balancing potentially conflicting concerns, such as quality-of-service, security and energy consumption. Many applications, not originally designed to support dynamic adaptation, are increasingly executed in environments that demand adaptation. The cost of rewriting these programs may be prohibitive, and even if such a task is undertaken, it may need to be repeated if the program is later required to adapt in a different way. How to introduce new adaptive behavior to existing, non-adaptive programs, while minimizing or completely avoiding direct modifications to the original code, is itself a challenging problem. The problem is further exacerbated if the application is programmed in a language that provides little or no inherent support for dynamic adaptation. This paper describes an approach to addressing this problem for C++ programs.

The predominant mechanism for implementing dynamic adaptation in object-oriented software is behavioral reflection [3, 10, 17, 18], which enables a program to observe its own behavior and make changes based on that behavior. Behavioral reflection, however, is not directly supported in many popular contemporary programming languages. In particular, Java provides only structural reflection, and C++ provides no reflection at all. Numerous approaches have been used to address this shortcoming, particularly in the case of Java. Examples include extending the language itself [1, 14, 25] or extending the Java Virtual Machine [21, 12, 19, 20, 8]. An alternative approach that has gained recent attention is to “automatically” augment the application code with support for behavioral reflection, in particular, code to intercept and redirect interactions among objects [7, 23, 24, 26].

In previous investigations [26], we used an aspect-oriented approach to add dynamic adaptation capabilities to existing applications. In the first step, at compile time, an “adaptation infrastructure” is woven into the existing program, making it adapt-ready with respect to one or more concerns. In the second step, at run time, an external entity (an adaptation administrator or another program) can use the infrastructure to introduce new behavior to the program. Recently, Sadjadi, et al. [23] extended this approach to construct TRAP/J, a generator tool that inserts partial behavioral reflection into existing Java programs. TRAP/J uses aspect-oriented programming (AOP) to create the adapt-ready program. Specifically, TRAP automatically generates aspects that replace instantiations of selected classes with instantiations of their corresponding “wrapper” classes. At run time, these wrapper classes provide hooks to enable interception and reification of method invocations targeted to objects of the selected classes.

The success of TRAP/J led us to wonder whether this general approach might be applied to C++. Since C++ programs are ubiquitous, the possibility of finding a means to transparently generate an adaptive version of an existing C++ program is appealing. To
achieve this goal, however, we had to overcome two major obstacles. First, TRAP/J uses Java’s structural reflection capabilities to reify inter-object messages into objects that can be manipulated and redirected to objects other than the intended receiver, but structural reflection is not available in C++. Second, TRAP/J uses aspects to weave in the adaptation infrastructure, but this approach cannot be directly applied to general C++ programs due to subtle differences in how Java and C++ objects are constructed.

Central to our solution is the use of generative programming techniques [6], which we used to create two components—a generator and a custom compiler—that together constitute TRAP/C++. Specifically, we addressed the lack of reflection in C++ by constructing a generator, TrapGen that automatically creates code artifacts that simulate the steps in the message-handling protocol. We overcame the weaving problem by using OpenC++ Version 2 [4] to generate a custom compiler, TrapCC, that produces the necessary wrapper classes from application classes. To evaluate TRAP/C++, we conducted a case study in which we enhanced an existing distributed C++ application so that new auditing capabilities can be added to it during execution. Such functionality can be useful to systems where downtime is prohibitively expensive; examples include systems to manage critical infrastructures (e.g., telecommunications, power grids, and financial systems).

The remainder of this paper is organized as follows. Section 2 describes the background information on the problem and prior work. Section 3 describes the TRAP/C++ approach. Section 4 presents the case study, and Section 5 reviews our findings and discusses possible future directions.

2. BACKGROUND

Approaches to dynamic adaptation in object-oriented systems tend to be built around a mechanism for modifying how objects handle messages. A general model for thinking about these mechanisms is Aksit’s composition-filters object model [2], which allows arbitrary objects to be adorned with functionality behind an interface that is plug-compatible with the application objects. At run-time, messages that are modified to use the adaptive protocol. Transient to the client, the invocation is actually made on the filter object, as depicted in step 1. The filter object responds by reifying the call into a message that is then sent to an AMH object (steps 2-3). A message comprises a method object paired with an array that contains the actual values that were passed as arguments to the method call. Method objects codify the signature of a given method, where a signature comprises the name of the method, the parameter types, and the return type. In this example, confi guration, one DMR, ad, is registered with the AMH. The AMH object queries each DMR to see if it has a method that is receptive to the message (step 4). In this example, ad reports that it can handle the message, and a call to invoke reifies the message and invokes it on ad (step 5). Since each DMR maintains a reference to the application object, when it is overriding an application method, it can optionally call the application method (step 6). Note that the “call back” operation invokes method invokeOrigMethod, which instructs the filter object to forward the message to the original application object, thereby circumventing any further adaptive handling.

In the case where no registered DMR can handle the message, the events proceed as in Figure 1 up until the AMH object queries each registered DMR (step 4). When all DMRs report that they cannot handle the message, the AMH communicates this information back to the filter object, who then invokes the call on the (original) application object. This outcome is the default case if there were no DMRs present. Notice that the design of this protocol makes extensive use of Java’s structural reflection, specifi cally the ability to reify method invocations into message objects and the ability to check the receptiveness of objects to messages. Because C++ does not support structural reflection, the protocol for adaptive message handling is more complex in C++.

2.2 Attaching Input Filters to Objects

The larger problem concerns how to attach input fi lters to objects of selected classes without requiring the application developer to modify the original code. In the original work, we addressed this problem using aspect-oriented programming [16] to weave the functionality of input fi lters around client-side calls to methods of objects of selected classes [26]. More recently, TRAP/J introduced the notion of filter classes that package the input-fi lter functionality behind an interface that is plug-compatible with the application class whose objects we wish to adorn with fi lters [22]. The “attachment” of input fi lters to selected objects is then implemented by systematically replacing client-side requests to instantiate selected classes with requests to instantiate the corresponding fi lter objects.

1This client-unaware redirection is achieved using techniques that are explained in Section 2.2.

2The defi nition of method signature that we use in this paper differs slightly from the one that some authorities use in that we do not consider the class name associated with a method to be part of the method’s signature.
class. Subsequent invocations of application-object operations are thus diverted to these filter objects. Once again, aspects are used to effect this replacement, but this use of aspects is much more systematic and indeed more focused than in the original approach.

Figure 1: TRAP/J Message-Handling Protocol

Figure 2 illustrates how TRAP/J augments an existing application with this adaptation infrastructure. The input is an existing Java application and a list of application classes, hereafter the selected classes, whose objects are to use adaptive message handling. From these inputs, the TRAP/J generator automatically synthesizes filter classes and the filter-attachment aspects that transform client requests to instantiate the selected classes with requests to instantiate their associated filter classes. These artifacts are combined with the initial application source and the generic AMH class and fed into AspectJ [15], which then weaves the aspects into the original application to create an executable that we shall henceforth refer to as the adapt-ready application.

Filter objects are plug-compatible with application objects because filter classes inherit from their associated application classes. In addition, filter classes contain references to an instance of the application class and an AMH object. Filter-attachment aspects replace instantiations of an application class with instantiations of the corresponding filter class by weaving so-called around advice around requests to allocate new instances of the application class off the heap. Note that this technique will not work for C++, which allows classes to be instantiated in automatic storage in addition to being allocated off the heap.

Independent of the generation of the adapt-ready program, is the development of DMRs. DMR classes are introduced at run time to provide new behaviors for application class methods. Each DMR class defines one or more methods that have signatures that are compatible with the methods of one or more application classes.

3. TRAP/C++

The objective of the current project is to gain a similar set of dynamic adaptation capabilities for C++ that we developed with TRAP/J for Java; that is, our intent was to develop TRAP/C++. As part of the TRAP/C++ investigations, we are exploring how input filters can be directly associated with application objects. This shift decreases the need for transformations of client-side code, and hence relies less on the use of aspects. This paper contributes a further refinement, which associates input filters with selected application classes without the need to weave logic into the clients of the adaptive objects. Thus, the approach presented here makes no use of aspects.

TRAP/C++ comprises a compiler and a generative-programming tool that cooperate to compile C++ programs into executables. Instances of selected classes in these executables handle messages using an adaptive protocol, similar to that depicted in Figure 1. TRAP/C++ differs from TRAP/J in two major ways. First, rather than using aspect technology to weave adaptive logic into a program, TRAP/C++ directly compiles selected classes into code that uses adaptive message handling (Section 3.1). Second, because C++ lacks the reflection capabilities of Java, an adaptive message-

3We refer to the objects referenced by a filter object as its objects.
handling protocol must be custom generated for each selected class (Section 3.2). This section also describes how adapt-ready application objects and DMR objects are made plug-compatible.

3.1 Selective adaptive message handling

Recall that the TRAP/J generator attaches input fi liers to objects of selected classes using attachment aspects, which systematically replace references to those objects with references to fi lier objects. Filter objects are plug-compatible with the objects they replace because each fi lier class extends the class of the object being replaced, over-riding all operations with methods that communicate with an AMH object. In contrast, TRAP/C++ does not distinguish application and fi lier objects. Rather, it attaches input fi liers by rewriting selected application classes into so-called wrapper classes, which combine the behavior defi ned in the application and fi lier classes. Rewriting occurs at compile time, after parsing but prior to code generation.

Figure 3 gives the elided code for class ClientSocket followed by the elided code for the wrapper class (also named ClientSocket) that is generated by rewriting. In this wrapper, the original receive method in the application class has been renamed to receive_Orig, and a new receive method is generated. This new receive method forwards calls to an AMH object. Because the wrapper class is generated at compile time and because it uses the same name as the application class, all client-program instantiations of ClientSocket will create wrapper objects rather than application objects. Notice that the original methods of the application class are still available in the wrapper class; their names have merely been appended with the _Orig suffix.

Application classes are rewritten into wrapper classes at compile time using a customized C++ compiler that we developed using Chiba’s OpenC++ meta-compiler [5]. OpenC++ is a fully functional C++ compiler that uses an extensible metaclass protocol. OpenC++ metaobjects encapsulate strategies for compiling different C++ language features, such as class declarations, assignment statements, and method invocations. Programmers create custom compilers by developing new metaclasses and then compiling and linking them with the open compiler. We developed a class metaclass called WrappableClass, which extends the built-in metaclass Class to rewrite class declarations into wrapper-class declarations prior to generating code. The resulting compiler, which we call TrapCC, uses metaobjects of class WrappableClass when compiling the classes selected for adaptive message handling and metaobjects of the default class (Class) when compiling all other classes. Selected classes are so designated using a metaclass declaration, e.g.,

```
metaclass WrappableClass ClientSocket;
```

which instructs the compiler to use the metaclass WrappableClass when compiling the declaration, defnition, and uses of class ClientSocket. Metaclass declarations are easily generated from the list of selected classes and do not require programmer intervention.

3.2 Customized message protocols

As mentioned in Section 2.1, fi lier objects in the TRAP/J protocol reify calls into message objects and then pass them on to an AMH object that can handle any kind of message. Because C++ does not support call reification, an AMH object must provide methods that correspond to the operations of the application objects that use it. Such methods have the same signature as these operations, with one additional parameter—a reference to the wrapper object, which is needed for DMRs that need to call back on the application object. Consequently, an AMH class defnition, and in-deed the entire adaptive message-handling protocol in TRAP/C++, must be custom generated for each selected application class.

Figure 4 depicts the adaptive message-handling protocol, customized for the ClientSocket class. Interaction begins when the client sends the message receive(buf,100) to cs. Because cs is a wrapper object, it then sends the message receive(buf,100,cs) to its adaptive message handler (amh). The message to amh contains a reference to cs, which is used to create a wrapper-adapter object csa (Step 3). These adapters convert calls of application-object methods into calls of wrapper-object methods. They are needed to allow DMRs, such as AuditorDelegate, to interact with a wrapper object as if it were an application object.

Step 4 checks to see if amh contains a delegate message receiver that can handle the message, and step 5 sends the message to the delegate. DMR objects are encapsulated by safe proxies, which contain the meta-information needed to check if a given delegate is receptive to a given message. In this example, ad is receptive to the receive message; so amh sends the message receive(buf,100,csa) to ads, which immediately forwards it to ad (Step 6). Note that the wrapper-adapter object (csa) is passed as a parameter of this message so that ad can invoke operations on the original application object if needed (steps 7 and 8).

3.3 TrapGen and DMR development

Code for the supporting objects that are used in the TRAP/C++ protocol (i.e., amh and csa) is automatically generated by a tool called TrapGen that we developed. Code for the delegate objects (i.e., ad and ads) must be provided by a developer, as they are specific to each type of desired adaptation. There are, of course, dependencies between hand-coded DMR classes and the classes generated by TrapGen. For example, while DMR classes are loaded at run time, an AMH object can only access instances of such a class using interfaces that are known at compile-time. Likewise, for a DMR object to invoke a method on an application object, some suitable interface for the application object must be known when the DMR class is compiled. One of the goals of the TRAP project is to investigate the usefulness of DMRs that handle messages of application objects of a variety of classes. Thus, a DMR interface cannot be inferred from the interface of a selected application class.

Currently, TRAP/J and TRAP/C++ programmers develop DMR classes using the interface of the original application class assuming that calls from a delegate back to the application object will not engender further adaptive message handling. We are aware of potential drawbacks in this assumption and note that it would be easy to generate a family of adapters, one for each operation in the application class, in which only invocations of methods of that type are converted to invocations of _Orig methods. To date, none of the examples we have encountered require this behavior.
or vice versa.

We address this problem by synthesizing the necessary interfaces from more fine-grained interfaces, called _application message receiver (AMR) interfaces_, which can be automatically generated from the declaration of an application class. Briefly, each operation in a selected application class engenders an AMR interface that declares exactly one operation, whose signature is identical to that of the operation in the application class. TrapGen also generates these AMR interfaces.

### 3.4 DMR development in TRAP/C++

DMR classes are implemented similarly to the approach used in TRAP/J (Section 2.2), with two notable differences. First, in order to be used with different types of application objects, DMR class methods must access these objects through AMR interfaces. Second, for each DMR class, the developer must also create a _safe proxy class_, whose methods allow clients to check if a given DMR object is receptive to a given message. We now explain these differences and their ramifications in more detail.

Suppose we wish to implement a DMR class, such as the class _AuditorDelegate_ from Figure 4. To be receptive to receive messages, objects of this class must provide a receive method, which may "call back" by invoking the receive method of the original application object (i.e., _cs_ in this example). Recall from Figure 4 that the call-back object (in this case the application-object adaptor _csa_) is passed as the final argument to the receive method of class _AuditorDelegate_. As mentioned previously, a given DMR object might be used to handle messages from different kinds of application objects. Consequently, a DMR class should not be designed to depend on a particular application class, which means the final argument to the receive method of _AuditorDelegate_ should not reference class _ClientSocket_ directly.

TrapGen supports the development of generic DMR classes by generating fine-grained AMR classes, each of which declares a single application-level operation (Section 3.3). Thus, the final argument to each method of a DMR class should reference an AMR class rather than a specific application-object class, such as class _ClientSocket_. Note that if a particular DMR method must call back using more than one of the original application object’s methods, then the call-back argument must implement multiple AMR interfaces. This case is easily handled by creating a derived class that inherits from each of the appropriate AMR classes. Accordingly, the final argument to the DMR method will then be a reference to this derived class. Note also that our decision to decouple DMR classes from application classes by designing DMR methods to depend on AMR classes alleviates the need to modify the original application classes to inherit from the generated AMR classes. DMR methods actually receive a reference to a wrapper-adapter object (e.g., _csa_ in Figure 4). It is this wrapper adapter that implements the AMR interface(s), and this class is generated by TrapGen.

Recall from Figure 4 that _AMH_ objects interact with so-called _safe proxy objects_ to pass messages to DMRs. Safe proxy objects provide a _hasMethod_ method, which allows clients (i.e., _AMH_ objects) to check the receptiveness of a DMR object to a given message. In addition, the safe proxy’s interface must be designed so as to make proxy objects plug-compatible with the corresponding DMR objects. Therefore, the developer must provide the DMR classes, each of which must be accompanied by a corresponding safe proxy class.

### 4. BROWSER SYSTEM CASE STUDY

To validate our work, we used TRAP/C++ to make an existing distributed browsing system adaptive. The browsing system comprises a server that streams text over a TCP socket, and a graphical client that receives and displays the text sent by the server. Our adaptation goal was to allow auditing software to be dynamically added to the client to enable monitoring of socket activity. The client program abstracts the functionality it requires from the C socket API (i.e., socket.h) with a class _ClientSocket_ (illustrated in Figure 5(a)). To accomplish our goal, we used TrapGen and TrapCC to make the _ClientSocket_ class adaptive, and developed a DMR class to provide the auditing functionality. For additional code excerpts, see the Appendix.
quired by OpenC++ to declare the metaclass of class ClientSocket to be a WrapperClass and another to include a generated header. Third, for testing purposes we added code to the client program to control the addition and removal of a DMR object. In future releases, this task will be unnecessary as DMR addition and removal will be controlled by an external agent. Finally, we used TrapCC to compile the client program code into an executable (resulting in the configuration shown in Figure 5(b)).

The second step to completing the case study was to develop a DMR class that implements the adaptation behavior. We developed a DMR class AuditorDelegate that defies a new version of the ClientSocket’s receive method. The AuditorDelegate::receive method is defined to write a time-stamped message to standard error and execute the original receive method for each call it receives. Additionally, we developed a safe proxy class that manages an AuditorDelegate object. This class implements an interface that is generated by TrapGen. In the future, there will be a program that will automatically generate safe DMR proxies and dynamic linking code based on the application and DMR classes. Finally, we compiled the DMR code into a shared object.

In conclusion, we found that, using TrapGen and TrapCC, we were able to transform a non-adaptive program into an adaptive client program capable of introducing new code at run-time to override methods of the ClientSocket class.

5. DISCUSSION

In constructing TRAP/C++, we have successfully ported TRAP/J to C++, and in doing so have developed a solution to address the lack of reflection in C++ problem and the weaving problem. For the reflection problem, we reconfigured the TRAP/J message-handling protocol to work without reflection by simulating the steps that require it. To address the weaving problem, we used OpenC++ to enable the original application source to incorporate adaptation logic. Automating TRAP/C++ through the creation of TrapGen and TrapCC has resulted in a convenient solution for making existing programs dynamically adaptive.

Through our work on TRAP/C++, we have made several findings. Overcoming the lack of reflection in C++ resulted in a solution with several implications. There is significantly more code generated by TRAP/C++ than by TRAP/J. Given a base program with N base-level classes that have M different method signatures among them, TRAP/C++ generates roughly 2M + N more classes than TRAP/J. However, DMR developers should benefit from the stronger typing afforded by the generation of AMR classes that declare the methods that a DMR developer will need to invoke. Suppose for example that a DMR developer forgets a parameter to a call back to an application object. In TRAP/C++, this DMR would not compile because the signature of the method called would not match that of the method in the generated AMR interface, but in TRAP/J the mistake would only be detected at run-time (e.g., step 6 in Figure 1) when the reflection message receives a malformed message.

In terms of performance, given a method call on an application object and the same call on a wrapped version of the object for which there is no DMR to handle the call, there are two additional calls associated with the wrapped version. It is difficult to generalize the impact these additional calls would have on a given system’s performance because it is dependent on the system’s usage patterns of the wrapped objects. The original and adapt-ready versions of the case study from Section 4 produced no noticeable differences in performance on the system we evaluated.

Several approaches have been developed in recent years that relate to TRAP/C++. Most similar is Hjalmtysen and Gray’s Dynamic C++ [13], which extends C++ with dynamic class reimplementation. However, their approach differs from ours in that a dynamically introduced implementation class is limited to implementing the methods of a statically defined interface class, whereas a DMR can override the methods of multiple application classes. Also, Dynamic C++ does not allow existing objects in a running system to be updated, but DMRs can override methods of existing wrapper class objects. Duffy, et al. [9] developed an approach for adding profiling to existing programs that utilizes the Decorator pattern [11]. This approach is similar to our wrapping technique, but does not address dynamic adaptation.

We are pursuing several exciting directions for future work on TRAP/C++. One is developing optimizations that reduce the amount of code generated per application class by TRAP/C++. Another is experimenting with building DMRs that are more closely associated with a given application class, and empowering those DMRs with greater ability to interface with the application while at the same time maintaining type safety. Finally, another future investigation is further automating TRAP/C++ by developing a front end for TrapGen and TrapCC.

6. ACKNOWLEDGEMENTS

We are grateful for the input and feedback provided by M. Sadjadi and other members of the Software Engineering and Network Systems (SENS) Lab on this project. This work is supported in part by NSF grants EIA-0000433, CDA-9700732, CCR-9901017, EIA-0130724, CCR-9912407, CCR-9984726, ITR-0313142, Department of the Navy, Office of Naval Research under Grant N00014-01-1-0744.

7. REFERENCES

APPENDIX

The following is an elided version of the code from the case study. The code illustrates how the ClientSocket class's receive method can be overridden by the AuditorDelegate.

Application class with the metaclass declaration:

```cpp
metaclass WrapperClass ClientSocket;
class ClientSocket : public Socket {
    public: bool receive(void*, size_t, int=0); ...
};
```

Wrapper class generated by TrapCC:

```cpp
class ClientSocket :
    : public Socket, public ClientSocketExt {
    public: bool receive(void*, size_t, int=0); 
    bool receive_Orig(void*, size_t, int=0); 

    // Methods inherited from Socket are copied/wrapped
    {return metaObj_->receive(buffer,bufsize,flags,this);} 
    ...
};
```

Classes generated by TrapGen:

```cpp
class ClientSocketExt { ClientSocketAMH *amhObj_; ...};
class ClientSocketAMH {
    public: 
        const char *addDelegate(const std::string &fileName, 
                             const std::string &className);
        void removeDelegate(const std::string &className);
        bool receive_Orig(void*, size_t, int=0);
        bool receive(void*, size_t, int=0, ClientSocket*);
    ...
};
```

... // One DMRMsgXRcvr for each app. method signature

```cpp
class DMRMsgXRcvr : public virtual AbstractDMR {
    public: virt...
};
```

```cpp
class AbstractDMR { // DMR safe proxy baseclass
    public: bool hasMethod(unsigned int id); ...
};
```

```cpp
class DMRMsg0Rcvr : virtual AbstractDMR {
    public: 
        virtual bool receive(void*, size_t, int, DMRMsg0Rcvr*)=0; ...
};
```

... // One DMRMsgXRcvr for each app. method signature
DMR classes created by the developer:

class AuditorDelegateSafe : public DMRMsg0Rcvr {
public:
    virtual bool receive(void*,size_t,int,Msg0Rcvr*); ...
private: AuditorDelegate *delegate_; 
};

class AuditorDMR {
public: bool receive(void*,size_t,int,Msg0Rcvr*); 
};

bool AuditorDMR::receive(void *buffer, size_t length, 
    int flags, Msg0Rcvr *adapter) 
{time_t t = time(NULL);
    std::cerr << "ClientSocket::receive called: 
    " << ctime(t) << "\n"
    return adapter->receive(buffer,length,flags);
}