G/MAX: An Object-Oriented Framework for Flexible and Efficient Multi-Protocol Communications

Donald F. Box
Tatsuya Suda

Technical Report 93-39

This material is based upon work supported by the National Science Foundation under Grant No. NCR-8907909. This research is also in part supported by University of California MICRO program.

G/MAX: An Object-Oriented Framework for Flexible and Efficient Multi-Protocol Communications

Donald F. Box and Tatsuya Suda

Department of Information and Computer Science,
University of California, Irvine,
Irvine, CA 92717-3425
(714) 725-3097 (voice)
(714) 856-4056 (fax)
dbox@ics.uci.edu, suda@ics.uci.edu

Abstract

This paper describes a new architecture for high performance distributed applications and a supporting framework. This architecture applies object-oriented design and implementation techniques to build a framework for platform-independent distributed application specification and implementation using existing programming languages and operating systems. It utilizes an efficient and extensible layering architecture that allows new abstract data types, new presentation-layer protocols, *and* new interprocess communication mechanisms to be added as they become necessary. Experimental results are presented demonstrating that the multiple layers of abstraction used do not compromise efficiency.

This material is based upon work supported by the National Science Foundation under Grant No. NCR-8907909. This research is also in part supported by University of California MICRO program.

I Introduction

Developing high performance distributed applications that must communicate with a diverse range of remote entities is a non-trivial task due to the complexity and heterogeneity of interapplication communication mechanisms and their interfaces. Adding distribution to existing applications can result in an inordinate amount of reengineering due to the lack of high-level support for distribution in most traditional environments. There has been considerable research in the areas of both distributed operating systems[1,2,3,4,5,6] that seek to provide distributional transparency to the designer, as well as programming languages for distributed application development[7,8,9,10,11] that offer language constructs to facilitate distribution. However these systems typically are not designed to interoperate with the wide variety of protocols or data formats that are used in a global internet environment.

The emergence of large scale high speed networks is shifting the performance bottleneck up from the communication substrate to the actual bandwidth offered to the user. This bottleneck is due largely to presentation formatting that must be performed to address heterogeneous internal data representation. [12] presents an architectural approach to reducing the impact of this processing based on Application Level Framing and Integrated Layer Processing. This approach benefits *not* from eliminating presentation formatting, but from (a) pipelining multiple processing stages to reduce unnecessary data movement and (b) allowing the application to process incoming data in application-specified units, thus reducing the penalty (at the presentation layer) for lost or misordered packets.

This paper describes the G/MAX system, a framework for high performance distributed application design and implementation. G/MAX is inspired by the concepts proposed in [12], and seeks to provide flexible and efficient *application-based* communication architecture. G/MAX provides language-level constructs and support for efficient and transparent distribution of application data objects. We are currently implementing and experimenting with this framework using the C++ programming language, which was chosen as the implementation language for its efficiency, its compatibility with the large body of existing C language code, and its support for object-oriented programming. The initial system is developed using traditional communication mechanisms (*e.g.*, sockets[13], Sun RPC[14], and XTI[15]), which will ultimately be replaced by the ADAPTIVE system[16] to allow finer grain control of the actual transmission of data objects.. This paper focuses on the Presentation layer of the system, and describes in detail the abstractions used for both presentation objects and communication medium convergence.

II Design Goals

The framework described in this paper is designed to satisfy the following goals:

Basic Encoding Rule Independence and Transparency: Due to variances in processor architec-

tures, operating systems, programming languages and their compilers, data (objects) that must be transported to a foreign host must be encoded into a format that all communicating parties can recognize. Under the OSI Reference Model[17], this task is the responsibility of Layer 6, the Presentation Layer. Presentation protocols are traditionally implemented using a *data definition language* for humans to specify the data formats to be exchanged. These data definitions are then compiled according to a set of *basic encoding rules* (BER) that specify the exact binary representation of each data type. For maximum flexibility, our framework allows Presentation Layer services to be transparently selected either for compatibility with existing applications (*e.g.*, ASN.1[18] for compatibility with OSI applications) or for the best match to new applications (*e.g.*, XDR with Multimedia extensions for bandwidth/processing intensive applications). A single object can be transported using multiple encoding schemes, with the correct one for a given end-to-end association selected automatically via strong typing and function/operator overloading. This allows a communicating entity to maintain a single internal data format that is most efficient for local processing, while using ASN.1 to communicate information to network management applications and a more efficient format for more time-critical communications.

Object/Task Location Independence and Transparency: The emergence of distributed object management systems and languages[19,8,20,21] has shown that using a distributed object-oriented paradigm is a powerful and expressive way to design and implement distributed applications. Our framework provides distributed object management facilities by augmenting objects with the necessary member functions to transparently or explicitly designate the location of data members and the constituent operations performed. The actual location of various application objects can be transparently selected by the application designer to match the communication characteristics of the application. Object location can be explicitly designated either at the time of object instantiation, or during the object's lifetime via a single member function. Finer-grain control of object location and migration can be specified by designating member functions or data members for remote invocation/instantiation, allowing the application designer to distribute a single object across multiple locations.

Communication Substrate Independence and Transparency: There is presently a very large number of communication substrates available to the distributed system designer (*e.g.*, TCP[22], OSI-TP4[23], VMTP[24], NETBLT[25], XTP[26], etc.), with new protocols on the horizon (*e.g.*, Bellcore's TP++, OSI HSTP). Each of these protocols provides varying levels of performance and types of service. To take advantage of advances in network technology, it is essential that applications sufficiently insulate themselves from idiosyncrasies of a given substrate without unduly reducing efficiency. Our framework seeks to decouple object transmission/reception from the underlying communication subsystem. This allows the same code to be used portably across many different communication subsystems without regard to the selected substrate. The frame-

work provides a minimal yet functional base interface to basic communication services, while allowing access to substrate-specific features, functions and formats in an efficient yet *isolatable* manner.

Efficient yet Robust "Higher Layer" Protocol Services: By designing the layering architecture for both transparency and efficiency, protocol layers which were previously considered bottlenecks in distributed applications can now be used in high performance systems. Studies have shown that presentation layer processing is a major bottleneck in network performance[12,27], due to both the complexity of the processing involved and the additional data movement incurred from translating data between formats. Our framework addresses both of these issues:

1. *Complexity* — The fundamental data types used in a presentation protocol are hand-coded and inlined to yield a highly efficient translation. Additionally, every built-in data type has a hand-tuned *type conversion* operator to allow efficient processing of built-in types. As composite data types are directly composed of the fundamental or built-in types, their translations are efficient as well. However, implementors are free to experiment with and hand-tune a given composite object's encoding and/or decoding.

2. *Redundant Data Copying* — The entire "data path" of the framework is designed to allow conversion-on-copy operations, scatter-read/gather-write, memory-mapped I/O, and "pipelining" of protocol processing operations. Additionally, we are experimenting with alternatives to the traditional socket interface to further reduce the need for copying.

Streamlined Development Process: Conventional systems require the designer to maintain a data description in a language other than the language being used to develop the application. Our framework allows designers and implementors to specify objects directly in the implementation language (*e.g.*, C++), without requiring an additional data description language or stub compiler. The fundamental data types and formats are precisely defined within the framework specification. This allows formats to be expressed unambiguously, while allowing the development cycle to be streamlined by using rapid prototyping techniques.

Medium Independence and Transparency: The class libraries used to encode data objects can easily be combined with the existing C++ `iostream`[1] class libraries currently being standardized by ANSI. This allows objects to be stored in a platform-independent format with no additional implementation effort. It also allows persistent objects to be "played out" over a communication channel by an application that is unaware of the underlying format simply by transmitting the contents of a file. This capability is also useful for debugging purposes, as an entire communication session can be captured to a file for later examination.

---

[1] The `iostream` library is the C++ analog to the C programming language's `stdio` library. However, it offers the advantages of being type-safe and extensible to encompass new data types and I/O devices

III Framework Components

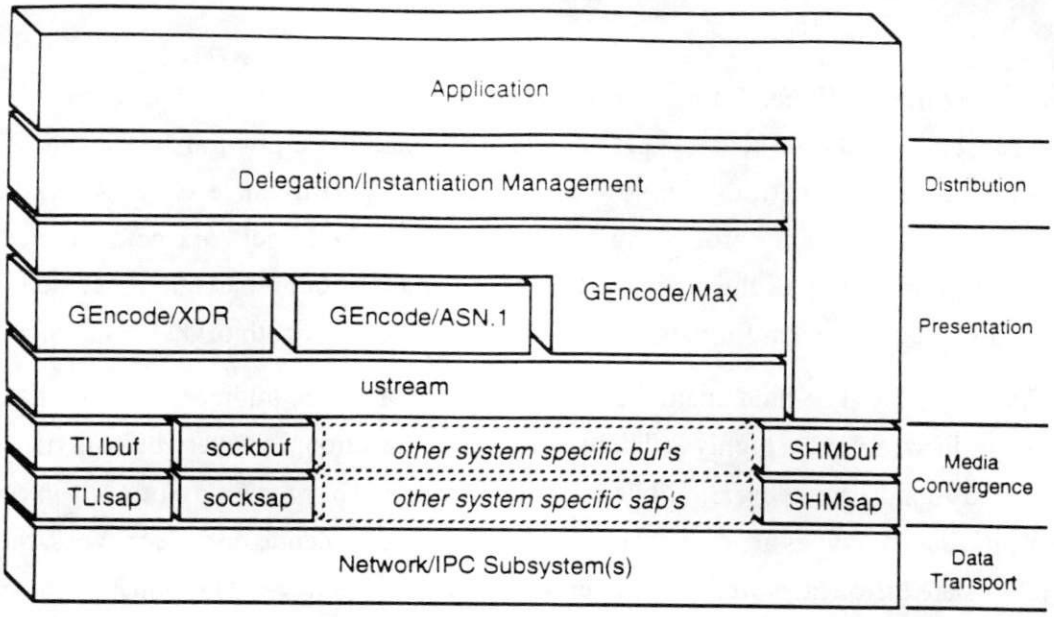

Figure 1: G/MAX Layering Architecture

Figure 1 shows the layering model used in our architecture. The **Data Transport** and **Media Convergence** layers correspond to the *Communication Substrate Dependent* level, as shown in Figure 2. The **Presentation** and **Distribution** layers correspond to the *Communication Substrate Independent* level, as shown in Figure 3. The following is a description of each layer.

Data Transport Layer

The Data Transport Layer provides the basic local and remote interprocess communication channel. This layer represents both the IPC mechanisms and their constituent Application Programmatic Interfaces (APIs). It is assumed that each underlying IPC mechanism provides either (1) a basic duplex data stream with either connection-oriented or connectionless semantics *or* (2) a shared memory interface with support for mapping a memory segment into and out of an address space.

Network Subsystems: The remote interprocess communication substrate. The basic connection-oriented network service is expected to provide an error-free, in-order delivery of bytes (*i.e.*, TCP[22] or equivalent). The basic connectionless network service is expected to simply provide a best effort delivery of datagrams (*i.e.*, UDP[28] or equivalent). Additionally, more diverse classes of network services can also be supported in this model. For example, ADAPTIVE[16] provides a multi-stream transport substrate that can be flexibly and adaptively configured to provide diverse grades of service to multimedia applications.

The Network Subsystem Layer also includes the APIs to network services, that allow user-space

applications to access data transport operations in a protocol independent manner. Several network APIs supported include the BSD sockets, System V TLI, POSIX XTI, ADAPTIVE API, *x*-kernel[29], NetBIOS[30], and the WINSOCK[31] library. Each of the interfaces provides both communication operations (*e.g.*, `open`, `close`, `send`, `recv`) and addressing/naming services (*e.g., address formats, name resolution and registration*).

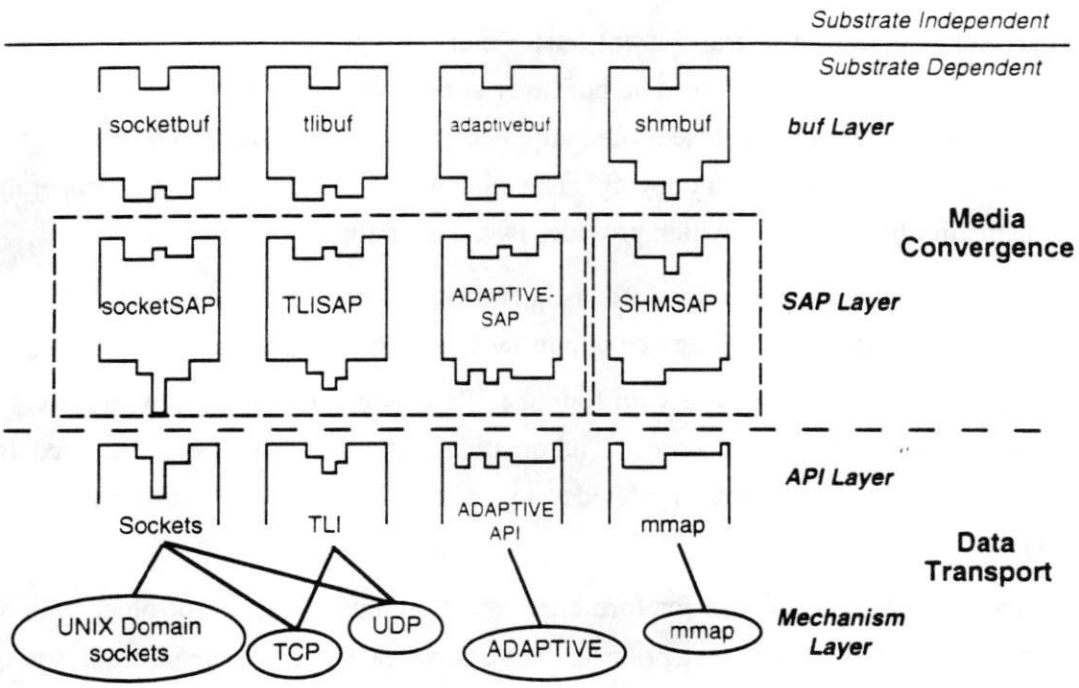

Figure 2: Communication Substrate-Dependent Level

Media Convergence Layer

The Media Convergence Layer provides a consistent, buffered interface to Data Transport services. It provides a basic data source/sink interface for higher layer subsystems, and uses the underlying Data Transport services to drain or replenish its internal buffering layer.

SAP Layer: The collection of uniform Service Access Points (SAPs) that provide a consistent interface to diverse interapplication communication services. Each SAP provides an *impedance match* between the native API provided by a given communication substrate and the basic communication service abstraction required by higher layer subsystems. There are two primary classes of SAPs, those based on duplex communication channels and those based on shared memory. SAPs based on duplex channels are required to support read, write, and connection management operations. SAPs based on shared memory must support basic attachment and detachment operations. Both classes of SAP must support a `SAPAddress` that can represent a communications endpoint for a given communication mechanism. This separation of addressing from basic communication operations allows a given communication mechanism (*e.g.*, TCP) to utilize several

possible API's (*e.g.*, sockets, XTI) and still maintain a single abstract SAPAddress format, thus decoupling the mechanism from the API. The uniformity of the SAP abstraction decouples the Communication Substrate from higher layer client subsystems. SAPs currently provided include `socketSAP`, `TLISAP`, `ADAPTIVESAP`, `SHMSAP` (Shared Memory via `mmap`). `SAPAddresses` include `TCPAddress`, `UDPAddress`, `UNIXDSAddress` (UNIX Domain Sockets), `ADAPTIVEAddress`, and `SHMAddress`.

buf Layer: The collection of transparent buffer managers that provide an efficient buffering scheme to the SAPs provided above. The buf layer is necessary to reduce the number of system calls needed to send a *composite* object (*i.e.*, an object with multiple data members) and to minimize the amount of redundant data copying. The buf Layer is based on and is interoperable with the C++ iostream library[32,33], which provides two sets of abstractions:

1. *streambuf* — the abstraction for a consumer/producer of bytes. To extend the `iostream` library to include a new I/O device or interface, one need only supply a streambuf interface to the device, and combine it with four specific classes via object composition[2] to allow existing classes to read and write to it automatically. The iostream library that accompanies the AT&T distribution of C++ provides `streambuf` interfaces to files and in-core memory.

2. *iostream* — the abstraction for formatted insertion and extraction of objects into/from a stream. The base classes `ostream` and `istream` each provide the insertion (output) and extraction (input) operators (<< and >> respectively) for each of the built-in data types supported by the language (*e.g.*, `char`, `int`, `float`). `istreams` and `ostreams` must be combined with a `streambuf` to provide a usable stream (*e.g.*, `istream` + `filebuf` = `ifstream`, a stream that *extracts* objects from a *file*). `iostreams` are not only extensible with respect to the devices they can support, but also with respect to the types of objects they can insert or extract. User-created data types (classes) can define their own input and output operations by *overloading* the insertion and extraction operators to support the new data type.

Each available communication subsystem has a corresponding SAP and buf that accesses its services (*e.g.*, `socketbuf`, `tlibuf`, `adaptivesap`, `shmbuf`). These bufs can then be combined with the standard `istream` and `ostream` classes to provide a *formatted* I/O channel, or with a new stream class (described below) to provide an *encoded* I/O channel.

---

[2] There are two primary mechanisms for object reuse, *compostion* and *inheritance*. Composition expresses *has-a* relationships, where object A contains an instance of object B. Inheritance expresses *is-a* relationships, where object A *is* a (specialized) instance of object B.

Presentation Layer

The Presentation Layer is responsible for resolving differences in data representations between heterogeneous host architectures. It accomplishes this by translating local internal data formats into an external format that can interpreted by the remote entity. It typically accomplishes this via one of two means:

1. *Explicit Typing* — each data object is tagged with a type identifier field that specifies the data type of the object in transit. It can then be followed by a length field that indicates the remaining number of octets (or fundamental data units). These two fields are then followed by the actual data octets. This is known as a T-L-V scheme (Tag-Length-Value) and is used as the basis for the Basic Encoding Rules of OSI ASN.1. This approach is in contrast to:
2. *Implicit Typing* — it is assumed that the receiver of the data is aware of exactly what type of data object is coming, therefore the tag field is redundant at best. This is the approach taken by XDR, a protocol that is designed to take advantage of regular data alignment and hardware-dictated formats.

[34] contains a comparison of three well known presentation protocols (XDR, ASN.1, and Apollo NDR). The authors resolve that T-L-V encodings are more general and potentially more bandwidth efficient, yet can be more complex to process, while fixed-format encodings such as XDR are more efficient to process, only slightly less efficient with respect to bandwidth, and can provide T-L-V functionality if necessary.

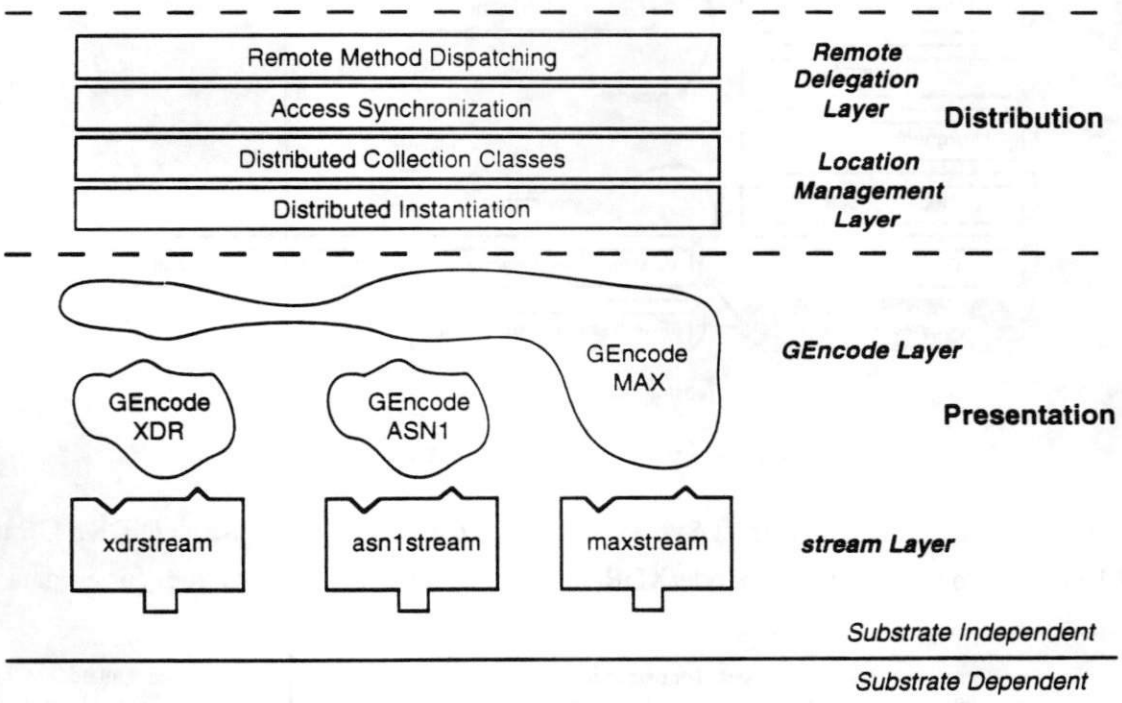

Figure 3: Communication Substrate-Independent Level

stream Layer: The stream layer is used to bind the various coding schemes listed below to an I/O channel (via its corresponding buf). For a given encoding scheme *Z*, an `izstream` and `ozstream` are implemented, providing *at least* the basic insertion or extraction operators for the built-in data types. Additionally, insertion or extraction operators will be provided for the corresponding GEncode class hierarchy that defines the basic data types used by the encoding scheme. In addition to providing the capability to statically bind an encoding scheme (stream) to an I/O device (buf) via object composition, *stream manipulators*[3] are provided to allow encoding schemes to be switched on the fly. By inserting (or extracting) a `z_on` manipulator into a stream, the previous formatting/encoding scheme is suspended and replaced with the *z* encoding. Inserting (or extracting) a `z_off` manipulator restores the original formatting/encoding scheme.

GEncode/XDR: GEncode/XDR is a class library of primitive base classes which correspond directly with the standard eXternal Data Representation (XDR)[35], as shown in Figure 4. This provides a set of Basic Encoding Rules that allow objects to be shared across diverse host platforms. For each *built-in* data type (*e.g.*, `char`, `int`, `double`) a type-conversion operator is provided to convert between language/compiler dependent types and formats to their corresponding GEncode/XDR base class. By leveraging off of C++ type management mechanisms, the presence of the GEncode layer can be completely transparent to the application programmer. GEncode also supports collection classes (*e.g.*, Lists, Dictionaries, Sets), C++ references, and pointers.

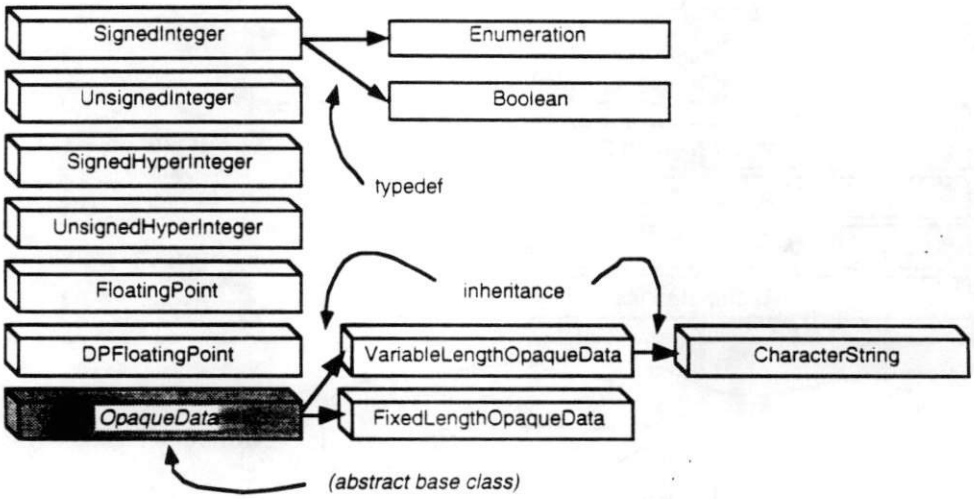

Figure 4: GEncode/XDR Class Hierarchy

GEncode/ASN.1: An OSI Abstract Syntax Notation One version of GEncode/XDR. GEncode/ASN.1 is more complex than GEncode/XDR, as it must address overflow issues for all data types

[3] Manipulators are "functions" that can be inserted or extracted into/from a stream. Inserting/extracting a manipulator has the effect of calling the manipulator's corresponding function with the target stream as the function's first argument. Manipulators allow otherwise complex encoding/formatting expressions to be written as a simple series of insertions/extractions.

(*i.e.*, reading an 8 octet integer into a 4 octet `long`). Also, it is difficult to directly support the ASN.1 notion of Set using only C++ constructs.

GEncode/MAX: A set of primitive base classes that represent Multimedia Activity eXtensions. These classes allow multimedia objects and basic application activities to be represented in a host platform independent manner. The Multimedia eXtensions we are currently implementing include support for

- 8KHz, 8 bit μ-law PCM audio
- 44.1KHz 16 bit linear PCM audio
- 44.1KHz 16 bit linear PCM audio (multi-channel)
- Indexed and Direct Color Pixmaps
- JPEG Still Image
- MPEG Motion Image

The Activity eXtensions we are currently incorporating include non-blocking and asynchronous remote procedure calls, C++ pointer-to-member-function semantics, and language-independent procedure name binding.

Distribution Layer

From the Presentation layer down, the support for distribution consists primarily of efficient mechanisms for copying objects to and from heterogeneous systems. The Distribution layer is the layer that creates an infrastructure for *transparently* migrating objects both with and without explicit initiation from the programmer. The application programmer can simply specify the location where the object should be located (if desired) and can then access the object as if it were located in the local address space. The Distribution layer consists of the following 2 sublayers:

Location Management Layer: The Location Management layer orchestrates the migration of objects based on both *explicit* (*i.e.*, the object's `existOn` member function is explicitly invoked) and *implicit* (*i.e.*, a member function declared as *remote* is invoked) events. Object locations are managed through the use of:

1. *Distributed Instantiation* — a technique where by overloading the language's `new` and `delete` operators, objects can be instantiated on remote hosts simply by passing an additional argument to the `new` operator. Calls to `new` without this parameter are routed to the standard `new` operator that allocates the object locally. The dereference operators (`->` and `*`) automatically dispatch member calls and accesses to the appropriate host (local or remote).
2. *Distributed Collection Classes* — a set of SmallTalk-like collection classes that allow a

collection of objects to be distributed across the network. *Distributed Iterators* are used to dispatch member function calls to multiple objects in a collection at once. Additionally, these classes can work in tandem with the ADAPTIVE system to perform prefetching of related objects to compensate high latency environments. When used with ADAPTIVE, these classes also allow application-based delivery of objects to the application (*e.g.*, potentially out-of-order, via upcalls, etc.).

Remote Delegation Layer: The Remote Delegation layer manages and execution of application object's member functions and arbitrates multiple accesses to a single object. By using a technique called *remote delegation*, an object's member functions are automatically invoked on the proper host system without programmer intervention. This is accomplished via two mechanisms: (1) overloading the dereference operators (`->` and `*`) to transparently dispatch member function calls, and (2) passing an additional `remote` parameter to the member function, which allows member functions to be executed on arbitrary hosts, not just the local host or the host the object actually exists on.

IV Object/Class Relationships

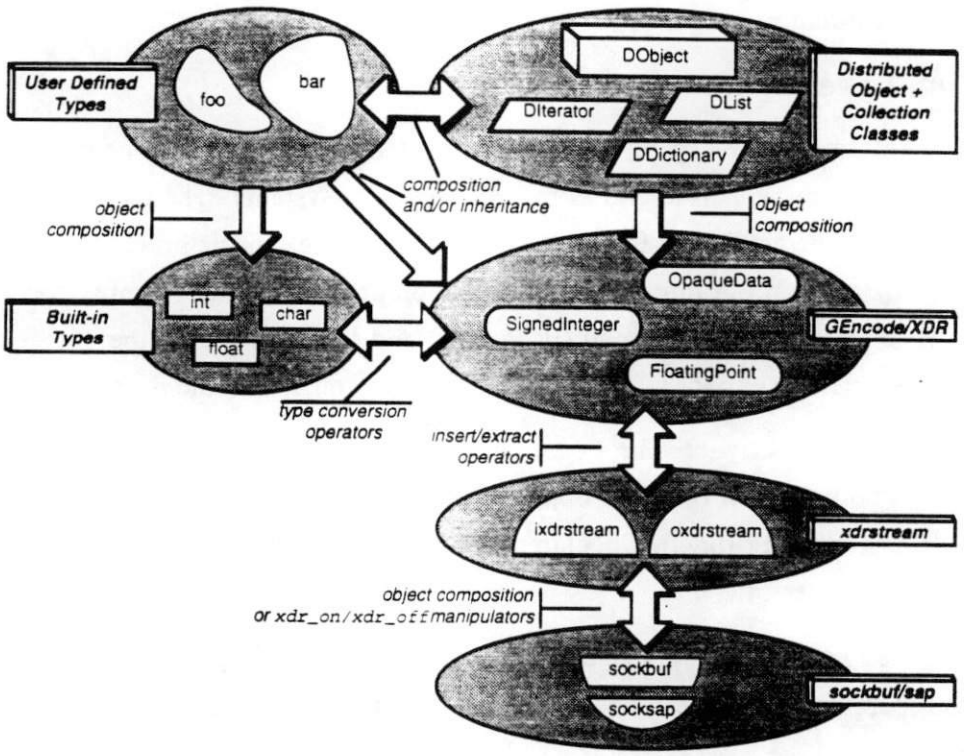

Figure 5: Object/Class Relationships

As the previous section describes, the G/MAX system consists of multiple hierarchies of C++ classes. This section provides concrete examples using the G/MAX system to illustrate the relationships among these class and object hierarchies. Figure 5 illustrates one vertical slice of the

system. The configuration shown uses the BSD socket facilities for IPC, and the Sun XDR encoding for presentation formatting. The `xdrstream` and `sockbuf` classes can bind the XDR protocol to the underlying communication channel either *statically* though composition and inheritance, or *dynamically* via the use of stream manipulators.

Figure 6 shows the detailed class and object hierarchy used to compose an `xsockstream`, a static binding of XDR to a buffered socket. The **has-a** relationships are via object composition, and the **is-a** relationships are via inheritance. The `sockbuf` uses its `socksap` data member as a uniform interface to the BSD `send` and `recv` calls. The `sockstreambase` acts as a virtual base class for all stream classes that will use `sockbuf` services. `sockstreambase` adds a consistent mechanism for error handling and exports the `sockbuf`'s connection management interface to derived stream classes. The classes `ixdrstream` and `oxdrstream` provide the insertion (<<) and extraction (>>) operators for both the set of built-in types (*e.g.*, `int`, `float`) and the set of GEncode/XDR (*e.g.*, `SignedInteger`, `FloatingPoint`) classes.

The classes just described are used in both the static and dynamic cases. To statically bind the XDR encoding to a buffered socket, the classes `ixsockstream`, `oxsockstream`, and `xsockstream` are used. `ixsockstream` is a class that is derived from both `sockstreambase` and `ixdrstream`. Using multiple inheritance, the new subclass inherits the behavior and interfaces of both superclasses (*e.g.*, the communication management is inherited from `sockstreambase`, and the XDR decoding is inherited from `ixdrstream`). It is this relationship that allows the encodings for a given data type to be defined exactly once (for `ixsockstream`) and to be inherited by all subsequent `ix*streams`, irrespective of the underlying communication mechanism. The `oxsockstream` is derived in a similar manner.

Finally to create a class capable of both sending and receiving XDR encoded data, the class `xsockstream` is composed of both an `ixsockstream` and an `oxsockstream`. It should be noted that for each `xsockstream` there is exactly *one* `sockbuf`. This is because the `sockstreambase` is a *virtual* base class[4] to the classes `ixsockstream` and `oxsockstream`.

---

[4] Virtual base classes allow multiple inheritance hierarchies to safely take the form of a DAG as well as a tree. If class A is a virtual base of classes B and C, and classes B and C are bases of class D, D will only have one instance of class A. If class A were a *non*-virtual base, objects of type D would have two instances of A, one for B and one for C.

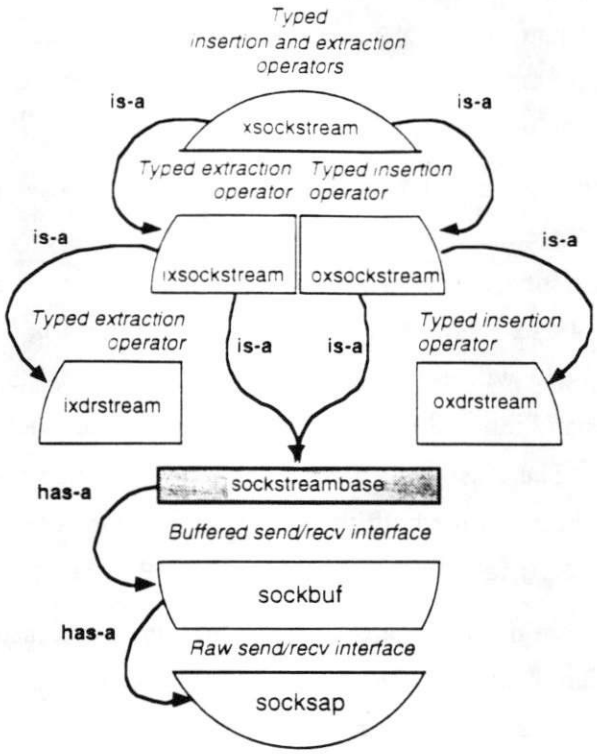

Figure 6: `xsockstream` Class/Object Hierarchy

Listing 1 demonstrates the steps necessary to encode a user defined type using xdrstreams. In the class declaration file (foo.h), one preprocessor macro declares the insertion and extraction operators and grants them *friend* status. As the operators are not actually member functions, friend status is necessary to access the class's private and protected data structures. The class definition file (foo.cp) requires a single macro to actually implement the insertion and extraction operators. Preprocessor macros are used to avoid the overhead of an additional preprocessor step (G/MAX macros are expanded by the standard C preprocessor). The form of the macros used is consistent enough to warrant a dedicated preprocessor, however, macros are used while the system is being developed to reduce initial development time. The use of macros also provides flexibility, as the programmer can choose to implement the insertion and extraction operators by hand, allowing selective transmission of each data member. Additionally, when used with the ADAPTIVE system, the programmer will be able to explicitly specify both what *sub-stream* a given data member will be transmitted on as well as what *mechanisms* will be used to send the member.

```
// foo.h
#include <xstream.h>

class foo {
private:
        long l;
        short s;
        char c;
        float f;
        double d;
        char buf[256];

        DECLARE_xdrfriends (foo);      this macro gives x_ops friend access
                                    and expands to:

//   friend oxdrstream& operator << (oxdrstream& ox, const foo& v);
//   friend ixdrstream& operator >> (ixdrstream& ix, foo& v);

public:
// ... additional member declaration
};

// foo.cp ////////////////////////////////////////////////////////////////
#include "foo.h"

IMPLEMENT_xdrfriends(foo,...)  // this macro implements the << and >> ops
//   the generated code is as follows:
//   oxdrstream& operator << (oxdrstream& ox, const foo& v)
//   {
//      return ox << v.l << v.s << v.c << v.f << v.d << v.buf;
//   }
//   ixdrstream& operator >> (ixdrstream& ix, foo& v)
//   {
//      return ix >> v.l >> v.s >> v.c >> v.f >> v.d >> v.buf;
//   }
// ... additional class member definitions
```

Listing 1: foo.h/foo.cp

Listing 2 demonstrates the use of sockstreams both statically and dynamically bound to the XDR format. The object `nstr` is a sockstream that uses the standard `istream` and `ostream` classes that accompany the standard `iostream` library. Objects sent via nstr would be formatted as ASCII text. The object `xnstr` is an `xsockstream` that uses the `ixdrstream` and `oxdrstream` classes that use XDR for data encoding. The use of C++'s strong typing and operator overloading allows the appropriate insertion or extraction operator to be selected based on the type of stream being used.

As `xnstr` is statically bound to XDR, all objects inserted (sent) to it will be formatted using XDR. To send objects via `nstr` using XDR, the `xdr_on` manipulator must be used to transform the type of `nstr` from `sockstream` to **x**`sockstream` . This transformation is temporary, and can be defeated either by an accompanying `xdr_off` manipulator or upon complete evaluation of the expression. Due the strong typing in C++, unbalanced `x_on` and `x_off` manipulators are detected at compile time. As `xnstr` is statically bound to XDR, no manipulator is necessary. The use of the `flush` manipulator in both statements is needed to force the stream to send the

contents of its send buffer.

```
#include <xstream.h>
#include <sockstream.h>

   create a TCP connection
sockstream nstr  InternetAddress ("london.ics.uci.edu", 3210));

   create an XDR TCP connection
xsockstream xnstr  InternetAddress  "paris.ics.uci.edu", 3210));

void f1 (void)
{
        foo red, green, blue;

        // send three objects to paris and flush the buffer
        xnstr << red << green << blue << flush;

        // send three objects to london and flush the buffer
        nstr << xdr_on << red << green << blue << flush;
}
```

Listing 2: sendFoo.cp

V Experimental Results

This section describes the results from a series of benchmarks of the system described in the previous section compared to several alternative solutions. The experiments were designed to illustrate the performance of the Presentation and Media Convergence layers described above. The first set of benchmarks measures the presentation formatting costs relative to memory bandwidth. The second set measures the end-to-end throughput when moving data between two hosts. Both sets of measurements were taken using Sun SPARCstation 2 workstations on an idle Ethernet.

All tests were performed using the class `foo` that appears in the previous examples. The first five data members were held constant, occupying 24 bytes when encoded. The sixth member ranged from a 16 bytes to 4K bytes and was represented as a fixed length opaque data object. This mix was chosen to exercise a variety of data conversions as well as allow some opaque data transfer.

Processing Costs vs. Memory Bandwidth

As has been described in this paper, there are many classes and objects that must participate in the transmission or reception of a given object. This series of measurements compares the processing costs of using `xdrstreams` to Sun's XDR library. Additionally, a version that simply copies an equivalent number of bytes of data from one location to another is included for comparison. The test program XDR-encoded an array of up to 1024 `foo` objects into a 1MB array of memory The decoding of the array was measured as well, and the results of encoding and decoding were averaged. The behavior of the test program is similar to the behavior of a multimedia application that

might read data from a codec and write/encode it to either the playout buffer or a network connection.

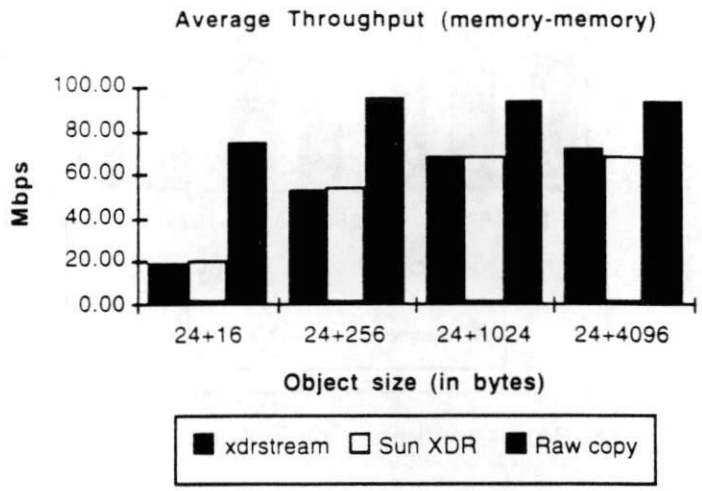

Figure 7: Processing vs. Copying Costs

Figure 7 illustrates the impact of formatting cost on throughput. When the data size is sufficiently large, the raw copy throughput approaches the memory bandwidth of the host. While both encoding mechanisms were less efficient than no processing at all, the relative performance of the two implementations varies by only +/- 6.3%. As the size of the opaque data member increases, the closer throughput of the two schemes approaches that of no processing whatsoever.

This result demonstrates that the processing costs incurred by the additional flexibility and abstraction of `xdrstreams` is only marginally greater than that incurred by the SunRPC library.

End-to-end Performance

To observe end-to-end performance, both the `xdrstream` and `sockbuf/socksap` classes are used. To reduce the system call overhead incurred per object transmission and reception, objects are passed to and from the kernel eight at a time. A larger number was not chosen to reduce the latency introduced by buffering. This required exactly one line of application code for the `sockbuf`, and was hand coded for the Sun XDR and raw cases.

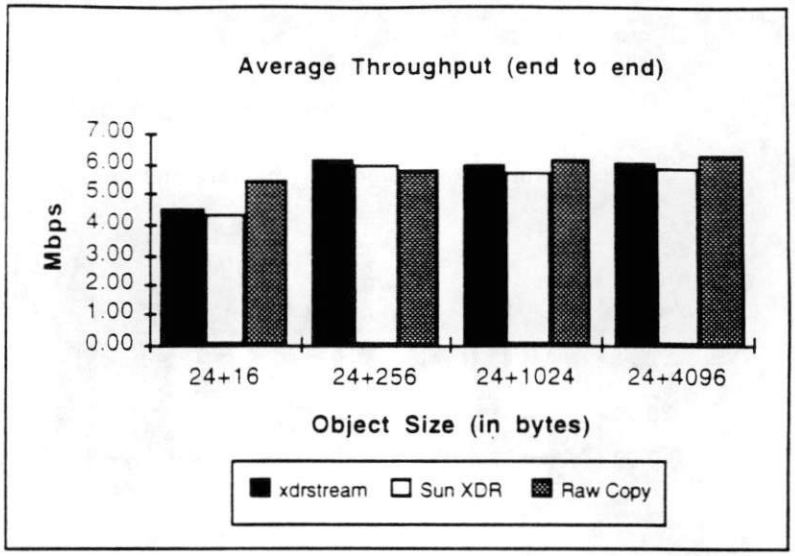

Figure 8: Processing Impact on End-to-end Throughput

As Figure 8 illustrates, the `xdrstream` outperforms the Sun implementation by an average of 3.4%. This is despite the fact that the Sun implementation was on average 1.2% faster than `xdrstreams`. This is due to the fact that the `sockbuf/xdrstream` combination are more tightly coupled than the hand-coded buffering scheme and Sun XDR, and thus (a) the compiler is able to make better optimizations, and (b) the behavior of the `sockbuf` is better suited to receiving partial data objects.

VI Conclusions

This paper has outlined a new architecture for the design and implementation of high performance distributed applications. We have verified experimentally that despite the additional flexibility they offer the application designer, the abstractions chosen for the Presentation and Media Convergence layers implemented thus far impose no performance penalty over production systems currently used. We are currently implementing the ADAPTIVE system as the Data Transport service, and are experimenting with a prototype of the Distribution layer described above which should offer higher application throughput than traditional techniques.

VII References

[1] E. J. Berglund, "An Introduction to the V-System," *IEEE Micro,* vol. 10, pp. 35-52, August 1986.

[2] A. S. Tanenbaum, R. V. Renesse, H. V. Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. V. Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM,* vol. 33, pp. 46-63, December 1990.

[3] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite

Network Operating System," *IEEE Computer*, vol. 21, pp. 23-36, February 1988.

[4] P. Dasgupta, R. LeBlanc, M. Ahamad, and U. Ramachandran, "The Clouds Distributed Operating System," *IEEE Computer*, vol. 24, no. 12, pp. 33-44, December 1991.

[5] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An Experiment in Distributed Operating Systems," *Communications of the ACM*, vol. 23, no. 2, pp. 92-104, February, 1980.

[6] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "Overview of the CHORUS Distributed Operating System," Tech. Rep. no. 90-25, Chorus Systems, 1990.

[7] J. Ahuja, N. Carriero, and D. Gelerntner, "Linda and Friends," *IEEE Computer*, vol. 19, no. 8, pp. 22-34, August, 1986.

[8] A. D. Birrell, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, vol. 13, no. 1, pp. 65-76, January, 1987.

[9] M. B. Jones, and R. F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," Tech. Rep. no. CMU-CS-87-150, Carnegie Mellon University, September, 1986.

[10] B. Liskov, "Distributed Programming in Argus," *Communications of the ACM*, vol. 31, no. 3, March, 1988.

[11] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Trans. on Software Eng.*, pp. 190-205, March, 1992.

[12] D. D. Clark, and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," , pp. 200-209, ACM, September, 1990.

[13] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman, "Design and Implementation of the 4.3 BSD UNIX Operating System," Addison-Wesley, 1989.

[14] Sun Microsystems, "RPC: Remote Procedure Call Protocol Specification," *Network Information Center RFC-1057*, June, 1988.

[15] IEEE, "Information Technology — Portable Operating System Interface (POSIX) — Part xx: Protocol Independent Interfaces (PII) (draft)," , 1992.

[16] D. F. Box, D. C. Schmidt, and T. Suda, "ADAPTIVE: An Object-Oriented Framework for Flexible and Adaptive Communication Systems," *in Proceedings of the IFIP Conference on High Performance Networking*, December, 1992.

[17] Intl. Organization for Standardization, "Information Processing Systems - Open Systems Interconnection - Basic Reference Model ISO 7498," 1984.

[18] Intl. Organization for Standardization, "Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)," May, 1987.

[19] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "A Distributed Implementation of the Shared Data-Object Model," *in Proceedings of the 1st USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19, IEEE, 1988.

[20] Object Management Group, "The Common Object Request Broker: Architecture and Specification Revision 1.1," OMG, December, 1991.

[21] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of the ARJUNA Distributed Programming System," *IEEE Software*, pp. 66-73, January, 1991.

[22] J. Postel, "Transmission Control Protocol," *Network Information Center RFC 793*, September, 1981.

[23] Intl. Organization for Standardization, "OSI Transport Protocol Specification ISO 8073," ISO, 1986.

[24] D. R. Cheriton, "VMTP: Versatile Message Transaction Protocol Specification," *Network Information Center RFC 1045*, February, 1988.

[25] D. D. Clark, M. L. Lambert, and L. Zhang, "NETBLT: A Bulk Data Transfer Protocol," *Network Information Center RFC 998*, March, 1987.

[26] Protocol Engines, Inc., "eXpress Transfer Protocol Specification," September, 1990.

[27] C. Huitema, and A. Doghri, "A High Speed Approach for the OSI Presentation Protocol," , IFIP, North-Holland, 1989.

[28] J. Postel, "User Datagram Protocol," *Network Information Center RFC 768*, August, 1980.

[29] N. C. Hutchinson, and L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64-76, January, 1991.

[30] IBM Corp., "NetBIOS Application Development Guide," April, 1987.

[31] Microsoft Corp., "Windows Sockets - An Open Interface for Network Programming under Microsoft Windows," June, 1992.

[32] Bjarne Stroustrup, "The C++ Programming Language," Addison-Wesley, 1990.

[33] AT&T, "The C++ Language System Library Manual," 1990.

[34] C. Partridge, and M. T. Rose, "A Comparison of External Data Formats," *in Proceedings of the IFIP TC6 Working Symposium*, IFIP, October, 1988.

[35] Sun Microsystems, Inc., "XDR: External Data Representation Standard," *Network Information Center RFC 1014*, June, 1987.