The Dissertation Committee for David Scott Page

certifies that this is the approved version of the following dissertation:

Effective Data Access in Software IO Frameworks

Committee:

Harrick M. Vin, Supervisor

Don S. Batory

James C. Browne

Michael D. Dahlin

Dilip D. Kandlur

R. Greg Lavender

Effective Data Access in Software IO Frameworks

by

David Scott Page, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2003

To Kazuko

Acknowledgments

While it is customary to acknowledge the professors, committee members, colleagues, friends, and family that helped bring the dissertation to fruition (and in my case the group is extensive and I do thank them), there were three people who have gone to such extraordinary lengths to support me that I would like to dedicate this space entirely to them.

I would like to thank my advisor, Professor Harrick Vin, who was instrumental in my building a foundation in systems and networking research from scratch. Harrick followed through with critical insights, continually prodding me to validate my abstract ideas, and challenging me by his example to stay focused on the problem at hand and strive for excellence. Along with academic and technical advice, Harrick supported me with employment as a GRA and RA for seven years, and was unfailingly kind and polite as he coaxed me past my innate stubbornness and towards the open–mindedness necessary for successful research. I have learned a great deal about conducting research from him and can only try to emulate his example and incredible achievements.

I would also like to thank Professor Greg Lavender, who, acting as my unofficial co–advisor, helped me refine my intuitions about my research. Greg coined the term "syntax directed binding," which succinctly captures the key idea of my dissertation. Greg was relentlessly positive about the value of my research, offering much needed encouragement when this process seemed interminable. Greg has also been very patient. Having hired me into his development group at Sun Microsystems, he then afforded me extensive time off to finish the dissertation and subsequently allowed me additional time to prepare for the defense. Like Harrick, Greg has deeply influenced me with his dedication and resulting achievements.

Finally, I would like to thank my dear wife Kazuko, who has not only tolerated this endeavor with extraordinary patience for the entire ten years we have been married, but has been a dedicated and doting mother to our three children — Lloyd–Ichiro (1999), Lionel–Shoji (2001), and Linette–Chifumi (2001) — largely organized and executed the daily operations of our home, all while financially supporting us by working full time. I am sincerely grateful for your help and apologize for taking so long. I dedicate this dissertation to you.

David Scott Page

The University of Texas at Austin
December 2003

Effective Data Access in Software IO Frameworks

Publication No. ____________________

David Scott Page, Ph.D.
The University of Texas at Austin, 2003

Supervisor: Harrick M. Vin

Advances in computer communications technologies have led to new classes of applications; these applications exchange data across networks using diverse data representations and encodings. While a computer programming language implementation intrinsically supports efficient access to data elements defined in that language, accessing (producing or consuming) IO data in its external exchange format, called a *transfer syntax*, requires specialized binding code to accommodate mismatches in the local and external data representations. Providing *effective input–output (IO) data access* means transparently accommodating these intra– and inter–layer syntactic complexities in order to extend to the programmer of IO data processing layers the convenience, efficiency, and accuracy of access to IO data automatically provided for programming language defined data.

This dissertation presents the theory, design, and implementation of a *syntax directed binding* facility that achieves effective IO data access. An improved programming practice introduces an abstraction boundary between the mechanisms of IO data syntax *navigation* and *access*, and the layer semantics or policies associated with the IO data *values*. The domain–specific language *Xyn* and its bit–level lexicon *Blex* succinctly specify the IO data syntax that defines the abstraction boundary. The Xyn compiler, *Xync*, codifies the implicit mapping between the labeled elements of a declarative Xyn/Blex specification and their presentation as identically labeled native programming language structures; i.e., Xync produces the binding code necessary to navigate and access the IO data syntax elements. Finally, an inter–layer optimization framework called *MetaXyn* exploits the *intra*–layer syntactic attributes exposed by Xyn to optimize the *inter*–layer composition.

The Xyn language has been used to specify the Internet Protocol version 4, and the Xync–generated binding code was evaluated in a modern software network router framework. We present and evaluate our results, and discuss our experiences.

Contents

Acknowledgments **v**

Abstract **vii**

Contents **ix**

Chapter 1 Introduction **1**

1.1 IO Data Binding 5

1.1.1 Challenges 8

1.1.2 Current Approaches 9

1.1.3 Open Issues 11

1.2 Layered Architectures and IO Data Binding 13

1.2.1 Challenges 14

1.2.2 Current Approaches 15

1.2.3 Open Issues 18

1.3 A New Approach 19

1.3.1 Syntax Directed Binding 19

1.3.2 Experimental Validation 23

1.3.3 Contributions 24

1.4 Dissertation Outline 25

Chapter 2 Architecture **26**

2.1 Intra–layer IO Data Access Architecture 27

2.1.1 Bindings 28

2.1.2 Navigation 29
2.1.3 Layered Bindings 31
2.2 Inter–layer IO Data Access Architecture 33
2.2.1 Meta–layer Protocol 33
2.2.2 Inter–layer Meta–protocols and Phases 35
2.3 Tool Architecture 35
2.4 Contributions 37

Chapter 3 The Xyn Language 38

3.1 IO Data Binding Languages 39
3.1.1 Conventional Approach to Transfer Syntax Binding 39
3.1.2 A New Approach to Intra–layer Binding 43
3.2 Xyn Language 44
3.2.1 Xyn Grammar 45
3.2.2 Why EBNF? 46
3.2.3 IO Data Parsing and Generation 47
3.2.4 Xyn Types 49
3.3 Blex: Bit–level Lexicon 54
3.4 Xync 56
3.5 Xyn and Blex Contributions 57

Chapter 4 Implementation 59

4.1 Binding Implementation Architecture 60
4.2 Implementing Blex Fields 62
4.2.1 Bit–access Layer 63
4.2.2 Translation or Interpretation or Encoding Layer 65
4.2.3 Host (User) Interface Layer 66
4.2.4 Layer Composite 66
4.3 Implementing Xyn Navigation Constructs 67
4.3.1 Fields and Semantic Variables 68
4.3.2 Aggregates 68

4.3.3 Views . . . 70
4.3.4 Choice . . . 72
4.3.5 Syntaxes, Xynions, and SubXynions . . . 73
4.3.6 Sequences . . . 75
4.4 Summary . . . 77

Chapter 5 Experience **78**
5.1 Click . . . 78
5.1.1 Why Click? . . . 79
5.1.2 Click Configuration . . . 80
5.2 Deploying Xyn in a Click Router . . . 80
5.2.1 Xyn Accessors in Click . . . 82
5.2.2 Code Metric . . . 87
5.3 Performance Evaluation . . . 87
5.3.1 Evaluation Environment . . . 88
5.3.2 Click Benchmark . . . 89
5.3.3 Xyn Feature Micro–benchmarks . . . 91
5.3.4 Insights . . . 97
5.4 Summary . . . 99

Chapter 6 MetaXyn **101**
6.1 Layout . . . 105
6.1.1 Contiguity . . . 107
6.1.2 MetaXyn Implementation . . . 108
6.2 Alignment . . . 110
6.2.1 Overview . . . 111
6.2.2 Intra–layer Metrics . . . 112
6.2.3 Inter–layer Metrics . . . 114
6.3 Non–Syntactic Considerations . . . 119
6.3.1 Mandatory Copies and Copy–like Operations . . . 119
6.3.2 Protection Boundary . . . 120

6.3.3 Access Mode and Persistent References 124

6.4 Layer Type 124

6.4.1 Type–safe Layer Binding 125

6.4.2 MetaXyn Implementation 126

6.5 Contributions 127

Chapter 7 Conclusions 129

7.1 Summary of Contributions 129

7.2 Research Directions 131

7.2.1 Implementation 131

7.2.2 Application Domains 133

7.2.3 Language Extensions 134

Appendix A ip.xyn: A Xyn Specification of IPv4 136

Bibliography 140

Vita 145

Chapter 1

Introduction

The past decade has seen the emergence of new classes of applications, including video conferencing, scientific grid computing, remote medical imaging, ubiquitous computing, and geographic information system (GIS) remote sensing. Successful deployment of an application in any of these classes requires exchanging data among application components distributed across a heterogeneous collection of networks and systems. The aggregate diversity of application communication requirements, and heterogeneity of networking and computing infrastructures, impose a complex — and sometimes conflicting — set of requirements on the data exchange mechanisms connecting the application components. Managing this complexity requires applying the software engineering principle of separation of concerns: specifically, factoring the aggregate communication requirements across a collection of functionally orthogonal service elements; a subset of which can then be composed into a complete communication service tailored to a particular application and its execution environment.

Inter–layer dependencies and implementation requirements, and the flow–oriented nature of application IO data exchange lead to a *vertically layered* service architecture: Each layer (service element) builds on an abstraction provided by its adjacent lower neighbor(s) and supplies a more specialized abstraction to its adjacent upper neighbor(s). In the computer networking domain, a service abstraction is specified in part by the structure of its exchanged IO data, i.e., the IO data's *type*. IO data type is defined using a *syntax*, which consists of the rules describing the possible constructions of exchanged IO data messages and how a message is mapped to the bit–sequence transmitted "on the wire." To implement a service, peer service layers (i.e., horizontal peers at opposing ends of the communications link) must share a common representation of the IO data they exchange. This encoding is called a *transfer* syntax. The transfer syntax also comprises part of a layer's interface with its adjacent lower neighbor. The interface to the layer's service is described by a *provider* or *user* syntax.

Conceptually, the IO data path forms a vertical cut through the service layers at a communications endpoint [Hufnagel and Browne, 1989]. Along the cut, each layer *imports* IO data arriving from the adjacent upstream layer into its own state, computes a new state, and then *exports* IO data to the adjacent downstream layer. (Note that downstream and upstream are relative to the data-flow; not the vertical organization of the IO data processing layers, which are typically presented so that IO data passes down through the layers in the case of output, and up in the case of input.) Importing (resp., exporting) transient IO data requires that the layer impose a type — a syntax — on the data content (resp., container) in order to *bind* program variables to the elements embedded in the data sequence that are to participate in the state computation. Binding IO data introduces significant complexities over binding programming language variables, and managing this complexity is critical to a robust IO data processing architecture and implementation.

Each entity in a program (i.e., state element or operation) has a name and is declared or inferred to be of some type, and in the course of program execution — before the name is referenced — the name must be *bound* to a location in the program's address space satisfying the constraints of its type. Once a binding is established, the location can be accessed through the entity's name, and its value becomes part of the program state. A typical programming language defines primitive types and operations on those types, and allows the programmer to define new types through the aggregation of previously defined types and definition of operations on the aggregation. The binding constraints imposed by a type include alignment, size, and layout; these vary among different system architectures. A binding to an element defined through a programming language environment mechanism, such as an automatically allocated variable on a stack or a dynamically allocated variable on a heap, is managed by the programming language environment.

In contrast to the simple and efficient access to *program defined* state elements provided by the programming language environment, binding program variable names to IO data is significantly more complex. This complexity arises from the transient nature and external definition of the IO data, and the resulting mismatches between a syntax's in–memory layout and that of the programming language type system.

An example is the fundamental layout conflict that arises due to programming language implementations being tailored to a target architecture such that a structure's layout incorporates inter-field padding in order to minimize access overhead at the expense of increased storage overhead. Conversely, a transfer syntax designer might choose the opposite space versus access–time tradeoff, applying techniques such as range–dependent widths of integer fields or variable–length entropy coded types, to exploit relatively abundant CPU capacity to accommodate relatively limited IO bandwidth. Similar techniques are possible using host programming languages that support defining types to sub–integral detail, at the expense of increased access overhead; however, established programming practice restricts this technique to the fitting of a program to some memory constrained hardware target; e.g., accessing a hardware IO device register. Additional complexities arise due to the heterogeneity of host architectures, since the layout of identically specified programming language types can differ due to the host architecture specific language implementations. Thus, a successful programming language mapping of an IO syntax on one host might be invalid on another host of a different architecture. Finally, an example of the tension between internally and externally defined constructs is the programming language implementation intentionally hiding details of an abstract data type from the programmer, while transfer syntax designers incorporate explicit structural information in the syntax for self descriptive types.

In addition to *intra*–layer IO data binding complexities arising from mismatches between a layer's programming language syntax and its implementation, complications also arise as a result of the *inter*–layer isolation critical to implementing the complex IO processing in the first place. Since the layers comprising the system and application interact with each other by exchanging IO data, and each layer imposes its own syntax on the data, an IO data layout resulting in minimal access overhead for one layer might be suboptimal for another layer. Receive (i.e., input) processing is predisposed to this problem since upper layers tend to have a more refined view of the data than lower layers. As a result a lower layer might deliver data with a suboptimal layout to an upper layer. Transmit (i.e., output) processing can suffer from an analogous problem. In this case, an upper layer might generate segments of data that are too long to be transmitted intact by a lower layer, resulting in fragmentation of the data and increased downstream processing overhead. Finally, there is a tradeoff between flexibility and type safety of layer compositions. If each layer treats its adjacent layers' data opaquely (i.e., ignores syntax information, allowing essentially arbitrary compositions) reuse is maximized, but at the expense of admitting invalid compositions. Conversely, requiring an exact match between adjacent transfer and provider syntaxes is overly restrictive since the expectation of the layered architecture is an upper layer implements a more specialized abstraction by building on its lower layer's more general abstraction, and each layer's transfer syntax reflects the layer's intended abstraction.

To summarize this brief introduction to the challenges of IO data access: accommodating the proliferation of application IO data encodings, including:

- *External Data Representation* (XDR) [Sun Microsystems, Inc., 1987], *Abstract Syntax Notation, One* and its associated *Basic Encoding Rules* (ASN.1/BER) [CCITT, 1988, ISO, 1987], OMG's CORBA *Internet Inter–Orb Protocol* (IIOP) [Object Management Group, 2002], and *MPEG–1* video [ISO/IEC 11172-2];

communicating across a heterogeneous networking infrastructure built from layers of protocols such as:

- *Internet Protocol version 4* (IPv4) [Postel, 1981a], *version 6* (IPv6) [Deering and Hinden, 1998], the TCP [Postel, 1981b], UDP [Postel, 1980], and RTP [Schulzrinne et al., 1998] *transport protocols*, and *Secure Sockets Layer* (SSL/TLS) [Dierks and Allen, 1998];

all implemented on heterogeneous system architectures featuring:

- CISC or RISC instruction sets, big– or little–endian integer byte ordering, and 32– or 64–bit integer registers;

has resulted in ad hoc structures in IO data processing programs. Fragile, low–level programming is required to access the externally defined IO data, while such programming is tedious due to myriad syntax fields. At the same time, static binding performance is demanded from an inherently dynamic process, while the required programming skills are unrelated to the application domain expertise.

Providing *effective data access in a software IO framework* requires transparently accommodating intra– and inter–layer syntactic complexities in order to extend to the programmer of IO data processing layers the convenience, efficiency, and accuracy of access automatically provided for programming language defined data. To this end, this dissertation presents the theory of *syntax directed binding*, in which effective IO data access is achieved by exploiting the regular structure that arises in IO data binding through a novel IO data access architecture that factors IO data access (i.e., binding semantics) from IO program processing (i.e., layer semantics). The dissertation contributes: 1) a declarative language *Xyn* in which to express the IO data syntax that succinctly describes the IO data access interface; 2) Xyn's compiler *Xync*, which codifies the mappings between the syntax structures and host programming language structures; and 3) *MetaXyn*, which breaks the per-layer encapsulation in a controlled manner in order to optimize the layer composition. These new abstractions simplify program development, improve type safety (both within a layer and in a composition of layers), and increase performance of modern IO–centric applications.

The remainder of Chapter 1 is organized as follows. Section 1.1 describes the challenges and current approaches to accessing IO data in software frameworks. Section 1.2 discusses the impact of layered architectures on IO data binding. Section 1.3 provides a more detailed overview of syntax directed binding, experience in its application, and a summary of its contributions. Section 1.4 concludes the chapter with an outline of the complete dissertation.

1.1 IO Data Binding

In order for the participants in a distributed computation to exchange information, the participants must share an understanding of the exchanged information's format or encoding. A *transfer syntax* describes the "on–the–wire" representation of the exchanged data, and is organized into sequences of *application* or *protocol data units* (ADU or PDU, respectively), each of which consists of an aggregate of fields.[1] The ADUs (and PDUs — ADU will stand for both in the sequel) correspond to logical entities in the program, such as an transaction request or portion of an image, while the *fields* comprising each ADU correspond to program variables or state of the computation. In order to generate or consume the exchanged data, the program variables representing the ADU fields must be bound to the corresponding fields in the transient IO data. This binding requires imposing the transfer syntax on the IO data buffer passed to or from the application, which, in turn, requires satisfying each syntax field's (programming language defined) type constraints. If the data is exchanged between hosts of different architecture (e.g., integer byte order "endianness"), or the programs are implemented in different languages imposing different rules for data layout (e.g., the alignment of structure fields and the resulting padding between adjacent fields), or the transfer syntax incorporates types that are not representable using native types (e.g., a constrained range of integers whose instances are represented using a minimal number of bits), then the syntax processing code must adapt these fields prior to their binding by program variables. Furthermore, when the application uses a virtualized service, such as a circuit or a file, the underlying service is not aware of these constraints, and hence, the data might need to be copied between the service's buffers and the program structures to achieve a conformant layout. Such adaptation requirements arise in both sending and receiving application or protocol data.

An example of a transfer syntax is the C language implementation of the Internet Protocol version 4 (IPv4) shown in Figure 1.1. The example shows some of the typical complexities arising from expressing a transfer syntax using a host programming language (that it is even possible indicates the low–level nature of C). This is a relatively simple example — streaming syntaxes such as MPEG1 video utilize a variety of specialized encoding techniques to minimize entropy in the stream, resulting in packed bit–sequences with variable–length fields falling on arbitrary alignment boundaries.

[1]While a transfer syntax describes the structure (type) of each message in an exchange, including alternatives of sequences of messages between a sender and receiver(s), a *protocol* constrains the ordering of message types (sequences) in a bi–directional exchange. Since protocol design and implementation is closely tied to the semantics of a layer, it falls outside the scope of this work; however, transfer syntax processing is an inherent (and factorable) part of protocol processing. Note that while the syntax of a protocol can be described using a (typically context free) grammar, describing the behavior of the protocol using a grammar might be impossible; for example, consider responses dependent on the values of received protocol (syntax) fields (terminals).

```
struct ip {
#if BYTE_ORDER == LITTLE_ENDIAN
        u_int   ip_hl:4,          /* header length */
                ip_v:4;           /* version */
#endif
#if BYTE_ORDER == BIG_ENDIAN
        u_int   ip_v:4,           /* version */
                ip_hl:4;          /* header length */
#endif
        u_char  ip_tos;           /* type of service */
        u_short ip_len;           /* total length */
        u_short ip_id;            /* identification */
        u_short ip_off;           /* fragment offset */
#define IP_RF 0x8000              /* rsvd fragment */
#define IP_DF 0x4000              /* dont fragment */
#define IP_MF 0x2000              /* more fragment */
#define IP_OFFMASK 01fff          /* mask fragment */
        u_char  ip_ttl;           /* time to live */
        u_char  ip_p;             /* protocol */
        u_short ip_sum;           /* checksum */
        struct  in_addr ip_src;   /* source address */
        struct  in_addr ip_dst;   /* dest address */
};

/*
 * Example access to ‘‘don’t fragment’’ flag
 */
struct ip *ipp = (struct ip *)m->data ;  /* IO data buffer  */
bool isDF = IP_DF & ipp->ip_off ;        /* import DF field */
ipp->ip_off = isDF ? ipp->ip_off | IP_DF /* export DF field */
                   : ipp->ip_off & !(IP_DF) ;
```

Figure 1.1: The Internet Protocol version 4 (IPv4) header (without options), followed by an example of accessing the single–bit “don’t fragment” field. Integer byte–order (preprocessor) macros are required to accommodate non–integral sized fields; in general, field widths are constrained by integral alignment requirements since crossing integral boundaries with bit–fields results in an undefined layout; field semantics are also encoded with macros (the fragment flags are shown, other enumerations are defined elsewhere in the header file); access to the IPv4 header aggregate requires an unsafe type–cast from a raw pointer (implicitly requiring integer alignment); finally, accessing a boolean (flag) field involves convoluted bit–level manipulations.

1.1.1 Challenges

The transformation between programming language data types and interchange format described by the transfer syntax is known as *encoding*, *marshaling*, *pickling*, or *serialization*, depending on the context. The following list summarizes the complexities found in this transformation.

- *IO data is transient and external to the processing program (i.e., service).* In contrast to straightforward read and write access to variables defined in a program, the IO data must be imported into, and exported from, the program. Importing and exporting IO data requires the program to dynamically impose a syntax on the memory range containing the data;

- *Complex and diverse IO data syntaxes and element types.* The IO data syntax can be complex, with an element's representation in a sequence dependent on its context in the sequence. Thus state is required in order to parse or generate the sequence. Furthermore, syntaxes are application or layer specific, with, for example, specialized syntaxes for remote procedure call argument marshaling and request–reply communication, and ISO 11172–2 (MPEG-1 video) streaming video sequences;

- *A mismatch between programming language type representation among application endpoints, and between an application endpoint and the external typing of IO data "on the wire."* Due to differences between host architecture representations of data types (even for a particular programming language), the binding must cope with differences among native host representations, including integer byte ordering and structure (i.e., type aggregate) inter–field padding. Some syntaxes specify a canonical "on–the–wire" representation, which might differ from that of the sender, receiver, or both. For example, most hosts implement a twos–complement signed integer, while a syntax encoding might use a sign–magnitude integer;

- *A mismatch between programming language defined data access mechanisms and IO data syntax access requirements.* Data types declared within a program are numeric and character types, or aggregates of such types with layouts optimized for fast access. The programming language environment provides intrinsic operators and storage management mechanisms. In contrast, IO datum (i.e., syntax element) encodings are typically optimized for space, and include variable length, entropy coded integer (i.e., minimal number of bits used on average), and padding–free aggregate representations. Furthermore, while the type of an element defined in a program is known from its declaration or inferred from its use, IO data encodings can use a self–descriptive encoding;

- *A mismatch between IO data syntax structures and programming language data organizational structures.* Decoding variant–type encodings requires dynamic discovery of instantiated type; typically involving explicit syntactic or semantic discrimination mechanisms not intrinsically supported by programming language constructs. Sequence encodings also utilize a variety of termination mechanisms, and are absent any prevailing conventions such as C's zero–terminated strings or the C++ STL container length field.

Code to perform the dynamic binding of IO data must be produced when a program is implemented or when the IO data syntax is extended; however, the low–level data manipulation and parsing techniques are not necessarily in the domain expertise of the implementer. Thus generating the binding code from an abstract specification has the potential of improving software performance and reliability.

1.1.2 Current Approaches

A variety of approaches to automate the coding of IO data bindings have been investigated over the past two decades. The general approach is to use a declarative language to abstractly specify interchange (IO data) types, along with corresponding tools to translate such a specification into bindings between some target programming language's structures and a transfer syntax's elements. Two widely known examples of this approach are the *External Data Representation* (XDR) [Sun Microsystems, Inc., 1987] and *Abstract Syntax Notation, One* and its associated *Basic Encoding Rules* (ASN.1/BER) [CCITT, 1988, ISO, 1987]. The XDR definition describes a canonical transfer syntax (i.e., on–the–wire representation) for C–like data types. The associated translation tool, *rpcgen*, takes an XDR description and generates the routines to translate between the C language representation and the XDR transfer syntax for each of the declared types.

In contrast to XDR's orientation towards application program data types, ASN.1 is intended to abstractly specify the transfer syntax used by layers implementing the OSI [ISO7498, 1988] protocol stack and applications using the stack. A tool architecture for ASN.1 specifications described in [Lavender et al., 1994] provides a mapping from the ASN.1 types into C++ classes, allowing the user to manipulate exchanged data within a conventional programming language environment prior to encoding and after decoding. It is open to transfer syntax specialization (e.g., ASN.1/BER or XDR can be used); however, this requires the user to implement the transformation rules for each ASN.1 type. While XDR and ASN.1/BER are designed as general purpose transformations for data exchange, other mechanisms are specific to some particular domain.

An IDL and translator specific to networking protocols is the Universal Stub Compiler (USC) [O'Malley et al., 1994], which constrains user types to a subset of C in order to generate binding code with performance equal to the hand–coded implementations of system–level communications protocols. In this case, the transfer syntax is the C–like, but padding–free, structures used to describe network protocol headers, such as those used in the TCP/IP suite [Postel, 1981a]. While the binding code handles issues primary to this domain, such as endianness (i.e., integer byte order) and idiosyncrasies of host–architecture structure padding, lack of support for the wide range of types found in presentation syntaxes limit its general applicability.

The Java serialization architecture [Sun Microsystems, Inc., 2001] utilizes a grammar to specify a canonical format for the serial representation of an object's type and values, including any object reference or primitive fields, and (recursively) super-classes. The default serialization mechanism queries Java's reflection interface to dynamically discover the object's structure, and then encodes the type information and field values in a sequence of octets according to the grammar. The default mechanism can be replaced or augmented by user provided routines. The coupling to the Java object model limits the generality of this approach.

An example of extending a familiar access idiom to IO data is the *socket++* library [Swaminathan, 1994], which encapsulates the BSD socket interface in a set of classes that export a C++ IOStream interface. This allows programmers to access the IO data (as an octet sequence) using a C++ *iterator*. Translations between user data types and the syntax implicit in the the underlying untyped byte–stream data available at the socket, can be implemented by overloading C++ iostreams input and output operators; however, the library does not include a facility to generate these, hence transfers are constrained to types encoded identically at all endpoints.

In addition to direct IO data access through transformations between host and external transfer syntax data, distributed application frameworks have extended various control flow idioms to the networked environment. Client–server and distributed–object programming model IDLs describe *remote procedure call* (RPC), *remote method invocation* (RMI), or *message passing interfaces* (MPI). IDL compilers augment the data transformation routines described above with control flow semantics such as *request–reply* or *message passing* through calls into a runtime library. For example, Sun's ONC [Srinivasan, 1995] implements remote procedure calls that utilize XDR to describe both the RPC parameters and the RPC protocol syntax. Similarly, Java RMI [Sun Microsystems, Inc., 1996] utilizes the serialization described above to pass parameters in method invocations across the Java Virtual Machine boundary. The Flexible IDL Compiler Kit (Flick) [Eide et al., 1997] investigates interposing abstract representations between stages of the IDL compilation process in order to enable application–specific RPC/RMI semantics; however, the intermediate language used to characterize an IDL is strongly influenced by its RPC/RMI heritage. Thus the abstracted data types and access idioms are those found in current RPC/RMI IDLs (e.g., procedure calls, attribute setters and getters), rather than a facility to extend support to other access idioms such as iterators over sequences of syntax elements, or transfer syntaxes such as MPEG–1 video or network protocol headers.

1.1.3 Open Issues

Each framework described above implements a transformation between host and network data representations that can be classified by a triplet consisting of: a specification language (typically declarative; sometimes implicit), a presentation (i.e., programming interface), and a transfer syntax. These transformation mechanisms can be operated independently from the control–flow (semantic) aspects of the frameworks; for example, the IDL translators and Java serialization framework allow not only the stand–alone use of the data type encoding facilities, but also the substitution of user–supplied encoding routines. Their decoupling of data encoding from invocation semantics suggests a general solution to specifying and implementing IO data encoding: in particular, one that allows abstract specification of the mapping between the presentation interface and the wire syntax, and generates the access mechanism to bind them. Such a solution could be utilized as a stand–alone binding generator, or incorporated into distributed application frameworks, such as RPC/RMI/MPI facilities, as part of the "back–end" of the framework's tool chain.

Why not XML? (I.e., Are Binary Protocols Still Relevant?)

One of the active areas of computer networking today is *web services*, which is essentially RPC using *extensible mark-up language* (XML) as the encoding for both the protocol and exchanged data. According to Dave Winer, one of the inventors of the technology, web–services "does not enable new applications" [Winer, 2002], instead it removes the dependencies on a particular vendor's platform (e.g., Microsoft's DCOM or .Net, or Sun Microsystem's Java). Although, a similar claim of interoperability has been made for preceding technologies, the proponents of web services point to XML as a truly vendor neutral language on which to base data exchange. While it is true that XML itself is an open specification, parsing XML data requires a *Document Type Definition* (DTD), or more recently an XML Schema — much like an RPC IDL defined using XDR. [2]

Since this dissertation focuses on binary (i.e., bit–level) encodings, it is important to clarify why the popularity of XML as a data interchange language does not render binary IO syntaxes irrelevant. Two key properties of XML make it unsuitable for IO–centric application data exchange. First, XML's <*attribute value*> mark-up style and text encoding make it extremely verbose. While this might not be as much of a problem for applications that exchange primarily textual (i.e., string) data, it is an ineffective way to encode large numerical or video data. Encoding large syntactic elements (e.g., an MPEG1 video slice) as a single attribute's value is still impractical, since the encoding would require a

[2]My personal observation is that the popularity of web services is largely due to its using HTTP as its transport, allowing it to pass through Internet firewalls opened to TCP/IP port 80 for WWW traffic.

final transformation between the MPEG wire format and XML's UTF–8 encoding [Yergeau, 2003].

Second, the overhead of parsing (or generating) XML data is high. Since the XML encoding is tree–based, the conventional parsing strategy (DOM) is to build the entire tree in memory, then traverse it looking for the desired attributes. A more recent parsing strategy (SAX) takes a list of attributes and traverses the tree, reporting each encounter with one of the requested attributes. While the latter strategy is more efficient for those cases where some subset of the XML document is of interest, parsing a large amount of XML data is still incurs significant overhead.

In summary, while web services based applications are clearly going to provide a significant (perhaps dominant) portion of networked application development, the proliferation of both data intensive applications and resource constrained consumer devices ensures a future for binary protocols.

1.2 Layered Architectures and IO Data Binding

In a general–purpose system, the software IO subsystem provides a framework through which an application can exchange data with its peers operating beyond the application's host execution environment boundary. In order to isolate applications from low–level details of system and communications link resources, the framework wraps the resources with an abstraction such as a *file* or a *virtual circuit.* The heterogeneity of application requirements has led to a proliferation of service abstractions, such as *remote procedure calls*, *file systems*, *name services*, *object request brokers*, and *transaction services.* The breadth and complexity of these service abstractions, and commonalities in their implementations motivates realizing complex service abstractions through composition of simpler services. This methodology observes the software engineering principle of *separation of concerns.* The nature of the services leads to a hierarchy of service layers in which each layer builds on the service provided by the layer below [Dijkstra, b,a].

The layers comprising an IO system service exchange data in the form of an untyped sequence of bytes; i.e., a layer does not attempt to interpret its neighbor's data; instead it treats it *opaquely.* For example, a filesystem provides a *file* abstraction (i.e., an uninterpreted sequence of bytes) to applications through the file read and write operations. It implements the file by segmenting the sequence across a collection of fixed sized blocks. The device controller for the disk provides a block interface to its users — e.g., the filesystem — and is oblivious to the contents of any block. Networking protocol layers are similarly isolated, where each layer *frames* a user's data by segmenting the data to conform with length constraints and adding a header and/or trailer to communicate with its peer, before passing it to the layer below. Opaque treatment of the exchanged data allows an IO system implementation to achieve a syntactic decoupling along the data path through each layer.

1.2.1 Challenges

While a layered implementation of the application and IO services is indicated by the software engineering principle of *separation of concerns*, several (non–semantic) conflicts arise due to the opaque treatment of data used to achieve isolation between the layers comprising a complete service or application:

- *Tension between isolation and performance in layer compositions.* Design isolation between layers results in an upstream layer choosing a locally optimized layout (including, for example, alignment and fragmentation decisions) for its exported data. This layout might require a downstream layer to copy this data in order to accommodate the type constraints of the variables through which it accesses the IO data. Excessive data copying inefficiently uses the memory hierarchy of modern architectures, reducing the overall system performance.

- *Tension between protection and performance in layer compositions.* The IO data path intersects an orthogonal layering in the virtual memory system at the boundary of the application's protection domain (virtual address space). Conventional implementations of this boundary crossing require a copy, contributing to the performance degradation of excessive copying.

- *Tension between isolation and type–safety in layer composition.* Conventional IO frameworks are implemented observing a layered architecture that maximizes flexibility by allowing the composition of essentially arbitrary layers, thus placing the burden of avoiding mismatched layer compositions on the system developer or deployer.

Since performance is critical to many emerging applications, the focus of much of the systems research of the past decade has been on delivering the exponential increases in hardware capacity to applications [Osterhout, 1990]. In the IO systems domain, two techniques have received extensive attention: reducing the isolation–performance overhead using *application level framing* (ALF) [Clark and Tennenhouse, 1990], and reducing the protection–performance overhead through a variety of user–system boundary crossing copy–avoidance techniques.

1.2.2 Current Approaches

Application level framing proposes that the data unit exchanged between application endpoints (i.e., an application data unit, or ADU) be *the smallest unit that the application (or presentation conversion function) can deal with out of order.* For ALF to be effective, the transport and lower layers need to observe the ALF boundaries when framing data units within their respective layers. When ALF is implemented end–to–end, the ADU is not segmented along the path to the receiver, and hence is delivered to the receiver's presentation processing layer in contiguous memory without additional copying overhead. This requires the sending application or presentation layer be aware of the portion of the path MTU available after the transport and below framing is imposed. While transport layers, such as those belonging to the TCP/IP suite, are able to discover the path MTU, at present, providing this to the presentation and application layers requires the use of *ad hoc* mechanisms (e.g., Unix *IO control* system call).

Since IO requires crossing the boundary of the application protection domain, IO data must be transferred across this boundary. Conventionally, this is accomplished by the read and write system calls copying the data between the application and system domains, but the copy overhead becomes a performance bottleneck when a large volume of data must be transferred. In response, a body of experimental systems research and numerous commercial operating systems enhancements have been developed to improve IO data transfer performance. These techniques can be classified as:

- Eliminating protection–domain boundary crossings by passing the IO data directly between two IO devices through *in–kernel data paths* [Fall and Pasquale, 1993] and the Windows NT *TransmitFile* system call [Microsoft, Inc, 1996], or bypassing the kernel by mapping network interface registers directly into application address space as done in *Osiris* [Druschel et al., 1994] and *U–net* [von Eicken et al., 1995].

- Reducing the cost of protection–domain boundary crossing through transient shared mappings of virtual–memory pages, examples of which include: *fbufs* [Druschel and Peterson, 1993], *container shipping* [Pasquale et al., 1994], *exokernel* [Engler et al., 1995], *transient copy–on–write* [Brustoloni and Steenkiste, 1996], and *IO–Lite* [Pai et al., 2000].

While each of these approaches has clearly demonstrated benefits in reducing IO processing overhead, the efforts to date have yet to deliver a compelling solution to managing systemic IO data processing overhead. Such a solution must address inefficiencies arising from conventional implementations of layered, general–purpose system architectures, without sacrificing the benefits of modularity and information hiding necessary for sound software engineering.

Eliminating the user–system boundary crossing by mapping device registers into the application space can reduce latency along with copy overhead; however, the technique is not an effective means for sharing a network link since it requires highly specialized hardware, can support a very limited number of applications requiring concurrent, mediated access to the device, and loses the security benefits of interposing system software on the path between the application and the external device. Eliminating the user–system boundary crossing by an in–kernel path is useful for delivering static content, such as file blocks or image data; however, it is not generally applicable to dynamically generated content, including techniques such as secure socket layer (SSL) encryption of application data.

While more appropriate for general IO system deployment than in–kernel paths or specialized hardware, copy–avoidance schemes that use virtual–memory manipulations must coexist with application level framing in order to be effective. Since the VM page granularity is typically 4KB or 8KB, and a typical path MTU is much smaller (e.g., two hosts connected by an Ethernet would have a 1500 byte path MTU "on the wire", and less at the application), the copy–avoidance protection boundary crossing effectively interposes a

VM page as a fixed size, mismatched, “transfer unit” into the path. Hence an effective IO system implementation can not implement these two techniques independently. Consider an application that interleaves generation of data into a buffer with write calls to transmit the buffer. If the VM manipulation is hidden within the write system calls, as is typically proposed, the application will either have to block until the buffer clears the physical link interface, or trigger a copy–on–write (COW) fault to preserve the queued contents. Analogously, an application receiving ALF data could accept frames at arbitrary addresses, and not necessarily in contiguous frames (e.g., out–of–order delivery and application data interspersed with network headers are both acceptable), which is a weaker constraint than that provided by the read system call. In summary, while a VM page mapping technique is an imperative for a performant IO system, current approaches need to be enhanced to propagate the nature of the user–system boundary crossing along the vertical cut through the IO processing layers. At the same time, the idiosyncrasies of the particular technique used need to be hidden from the layer developers.

Another potential contributor to copy overhead is accommodating the intra–layer binding requirements. If the layout of the IO data delivered to a layer (e.g., the output of some physical device or result of an upstream computation) does not conform to the alignment and contiguity requirements of the host types that will be bound to the syntax fields imposed on the IO data, then the data must be copied to a suitable layout. By negotiating an optimal alignment and contiguity for the delivery of IO data to a domain, the collection of layers that will access an IO data buffer can minimize this source of copy overhead. At present, layers are implicitly, if at all, designed to avoid alignment induced copies. For example, Ethernet device drivers utilize DMA buffers aligned on a two–byte displacement from a four–byte alignment boundary in order to deliver the payload beyond the 14–byte Ethernet header on a four–byte boundary. User–level application programs exploit the copy across the user–system boundary in order to align data properly for subsequent processing. One system, the Click Modular Router [Kohler et al., 2000], utilizes a table of per–layer alignment requirements in conjunction with a data-flow graph to compute when a “copy” element must be inserted into a composition of layers; however, the each message is copied in its entirety (rather than lazily copying only the segment to be accessed by the layer), and alignments are assumed to be fixed (rather than attempting to negotiate delivery of the data on an optimal alignment).

Conventional IO frameworks allow the composition of essentially arbitrary layers. While such flexibility simplifies the development of layers and the framework, the burden of ensuring valid compositions falls on the deployer. Since, effectively, layers are bound through their exchanged IO data, testing for compatible syntaxes at layer interfaces could contribute to improved type safety. In general, this is a difficult problem due to the layer isolation achieved by a layer's syntax being opaque (i.e., a byte sequence); however, current systems do provide some degree of checking. The IP layer and its associated transport protocols share information through an IP "pseudo header" that includes the IP protocol ID for the transport layer, implicitly ensuring at runtime that the appropriate transport layer was bound to a particular IP demultiplexer port. In contrast to deducing type information from the syntax content, the *gstreamer* Multimedia Framework [Walthinsen, 1999] tags each layer (called plug-ins in that architecture) interface with a MIME type, and tests for compatible types when layers are composed.

1.2.3 Open Issues

In general, IO data path optimization techniques require sharing information among components. If this is done by exposing the implementation of a component, it destroys the benefits of information hiding achieved through conventional layered system and IO architectures (namely, that layers can be implemented independently). Instead, what is needed is a resolution of the tensions arising in layered compositions that can break the layer encapsulation in a controlled manner, retaining the benefits of modularity and information hiding achieved by the conventional architecture. Given the continued importance of effective IO processing, the resulting challenge is how to incorporate the lessons and techniques from the existing body of work into future software–system architectures and implementations, and still maintain the integrity of the layered IO system architecture.

1.3 A New Approach

Each of the state–of–the–art system and application IO data processing framework approaches discussed in the previous two sections solves some of the items comprising the intra– and inter–layer binding challenges, listed in their respective sections. Unfortunately, none is a general solution, in the sense that each is constrained along one or more of the dimensions of access idiom, transfer syntax, or layer composition support. To a large extent, this limitation arises from intertwined intra–layer IO data access and layer semantics. Furthermore, none of the current approaches exploit the knowledge available about type requirements for binding *within* a layer to improve bindings *between* layers; either to improve type safety or enhance performance through copy avoidance. As a result, developers of IO programs must use ad hoc methods to implement an unsupported access idiom or transfer syntax, or to meet performance requirements.

1.3.1 Syntax Directed Binding

To remedy what is essentially a problem of insufficient abstraction — specifically, the need to factor the complexities of IO data access (i.e., the binding semantics) from program semantics — this dissertation describes an architecture and implementation of a facility for effective IO data access. A developer specifies an IO processing layer's user and transfer syntaxes in the domain–specific *Xyn* language. In general, the motivations for creating a domain specific language are succinct expression and masking some underlying complexity. In the case of Xyn, a specialized declarative syntax succinctly captures the on-the-wire structure of the IO data bits. At the same time, by mapping constructs in the Xyn meta-syntax to structures in the host programming language, a Xyn specification implicitly defines the IO data binding semantics; i.e., the actions necessary to import IO data for processing in a program layer, and to export the layer's results to the next layer. This relationship forms the basis of *syntax directed binding*, which isolates the IO data processing program developer from the extensive collection of low–level details required to tie the layer's semantic actions to the transient IO data on which they operate. An example of a Xyn specification and usage is given in Figure 1.2.

In contrast to the conventional IPv4 implementation shown previously in Figure 1.1,

```
syntax IPv4Net ;
IP  // Protocol layer processing view
  : // (fixed header only).
  ip_v   : 0b0100     // IP version 4
  ip_hl  : UIMSBF<4>  // header length...
  ip_hl_bytes : {ip_hl * 4 }// ...in bytes
  ip_tos : TOS        // type of service
  ip_len : UIMSBF<16> // total length
  ip_id  : BSLBF<16>  // identification
  ip_RF  : 0b0        // reserved (flag)
  ip_DF  : Boolean    // don't fragment
  ip_MF  : Boolean    // more fragments
  ip_off : UIMSBF<13> // fragment offset
  ip_ttl : UIMSBF<8>  // time-to-live
  ip_p   : Protocol   // user protocol
  ip_sum : UIMSBF<16> // checksum
  ip_src : Address    // source address
  ip_dst : Address    // dest address
  ;

//
// Example access to the don't fragment (DF) flag
//
IP & ip( IP::MakeAccessor( m->data ) ) ;
IP * const ipp( &ip ) ;   // mimic pointer access
bool isDF = ipp->ip_DF ;  // import syntax field
ipp->ip_DF = isDF ;       // export layer state
```

Figure 1.2: The Xyn Specification of an Internet Protocol version 4 (IPv4) header (without options), consisting of: IPv4Net, the syntax declaration; IP, the top-level message structure and syntax entry point; An aggregate of header fields including primitive field types and references to Xyn types defined elsewhere in the module; a semantic variable specifying the header length in bytes, as derived from an encoded header length field; and an example of accessing the ip_DF field as a native type.

the Xyn specification of IPv4 succinctly describes the external representation of the syntax, rather than its low–level implementation. The declarative specification can be translated into an implementation through a meta–syntax that maps IO data syntax constructs to host types. This mapping allows transparent access to encoded IO data fields by presenting them to the layer implementer as native host types. For example, note how the "don't fragment" field can now be treated identically to its corresponding native boolean type.

Xyn's (meta–)syntax is specified by a EBNF–style, context–free grammar, designed to accommodate transfer syntaxes as diverse as those of XDR [Sun Microsystems, Inc., 1987], ASN.1/BER [CCITT, 1988, ISO, 1987], MPEG–1 video [ISO/IEC 11172-2], and the TCP/IP protocol suite [Postel, 1981a]. Xyn's terminals are from a rich, extensible bit–level lexicon called *Blex*, that includes, e.g., a 5–bit sign–magnitude integer or a variable–length entropy coded table index. The Xyn translator, called *Xync*, compiles the syntax specification into a collection of types and access idioms expressed in the host programming language environment (currently C++), where a sign–magnitude integer embedded in the IO data syntax is accessed as a native two's–complement integer and an entropy code as a reference to a table entry holding the encoded value.

Access idioms provide the interface to the syntax fields, and include assignment and pointers to primitive and aggregated types, procedure call arguments, and sequence iterators. These access idioms are then utilized through one of the two *navigation* styles:

- *Semantically driven.* Provides random access to syntax fields, which is necessary to process protocol headers such as those of TCP/IP. Inherently exploits late binding (i.e., postponing encoding or decoding of IO data until it is needed for transmission or consumption);

- *Syntactically driven.* Sequential translation, in which the generator or parser controls the IO data processing. This style is well–suited to presentation syntax processing; particularly when there are variable length fields or sequences of elements, or interpretation depends on earlier fields, since either precludes efficient random access.

Random access is particularly useful in processing a layer decomposed into functionally orthogonal computational elements, where each element independently operates on a subset of syntax fields. This navigation style is well suited for operations such as partial protocol processing performed by an IP router. Conversely, sequential access provides a uni–directional translation between a monolithic block of IO syntax and the program state, necessary for highly compressed syntaxes such as MPEG–1 video.

Along with intra–layer access generation, the facility includes MetaXyn, which supports inter–layer techniques applicable to a sequence of layers composed to process an IO data flow: copy avoidance and type safety improvements. Since, effectively, the layers are bound through the passed IO data, the syntax information necessary to implement the inter–layer techniques can be collected during the processing of the intra–layer binding specification. For example, avoiding copying within a layer requires delivering IO data that meets the alignment requirements of the compiled intra–layer syntax element accessors. A syntax such as TCP/IP [Postel, 1981a] is designed so that fields fall on "natural" host alignment boundaries of one, two, or four bytes, provided the IO data is delivered on a four byte address boundary. An optimal alignment can be deduced from the Xyn specification by fitting the specification to the small set of possible alignments and evaluating the access costs for each alignment.[3] The inter–layer optimization is implemented by pushing an analysis of the aggregate message type upstream to any point along the IO path where a copy–like[4] an operation is required. For example, the boundary between a network interface and the system memory is a physical boundary, which mandates a copy operation. This copy should be exploited to optimally locate the data in memory for subsequent accesses until the IO data reaches the next copy boundary.

A further benefit of exporting syntactic constraint information from each layer, is that it allows a degree of syntactic type checking (equivalent to that provided by a conventional procedure call interface, if each syntactic element was a parameter of the call). The facility enhances the type–safety of the composite, by comparing the IO data syntaxes of the adjacent layer interfaces — that is, the syntax exported by some layer (that producing the IO data) and the syntax imported by the adjacent layer (that consuming the IO data). In spite of the architectural model of passing an opaque sequence of bytes between layers, often there is more extensive type information available. For example, the TCP and UDP transport layer protocols exchange a "pseudo–header" with the IP inter–network layer protocol, consisting of a subset of the IP header fields such as the source and destination addresses. While the pseudo–header content is passed in various ways in current implementations, it could be made part of the provider (upper) syntax of IP, and the transfer (lower) syntax of TCP and UDP, and passed as a prefix to the transport layer header. In this manner, an error such as attempting to bind an IP version 6 instance of TCP to an IP version 4 instance of the IP layer would be caught by the mismatched pseudo–header syntax.

Together, the components Xyn, Blex, Xync, and MetaXyn constitute a useful facility

[3]While this algorithm has high computational complexity, its naivete is ameliorated by the limited set of possible alignments (typically, a maximum of 8), and being computed statically (i.e., offline) per layer.

[4]This could be a copy, or a transformation of (most of) the data.

for a critical aspect of IO program development and deployment. The IO data access facility is designed to be extensible, in order to accommodate new access idioms and transfer syntaxes; open, to coexist with or be incorporated into program development and runtime frameworks; and portable, to allow deployment on a wide range of platforms.

1.3.2 Experimental Validation

The Click Modular Router Toolkit [Kohler et al., 2000] provides the infrastructure for validating the dissertation's approach. Click implements the IP layer and related protocols such as the *Address Resolution Protocol* (ARP) [Plummer, 1982] through a collection of simple–purpose elements that are combined into an IP router per a graph specification written in the Click configuration language. Each Click element bound into the router operates on some portion of the syntax, and hence must dynamically bind its state to the appropriate syntax fields. A Click router performs as well as monolithic software routers implemented in current operating systems, and so provides a legitimate baseline for a performance evaluation as well as a functional evaluation of the dissertation's approach.

The initial step in the validation was to specify the IP network (i.e., transfer) syntax, including a representative sample of the IP options, using the Xyn meta–syntax language. Next, the Xyn translator, Xync, was used to compile the Xyn specification into a set of access mechanisms coded in C++ and written to a header file. For several Click elements along the data path — chosen for their disparate IO data access characteristics — the generated header file was substituted for the Click header file containing the conventional C–structure and pre–processor macro definitions of the IPv4 protocol syntax. At the entry point for each module under test, a Xyn IP header binding object instantiated with the passed in IO data buffer was substituted for the conventional type-cast of the IO data buffer address to a C structure pointer. Through the use of the C++ template facility, and overloaded assignment and type conversion operators, the generated access mechanism replaced the C structure field references without further editing of the Click element description, although one module was simplified by exploiting the more abstract interface to the IO data syntax.

The updated modules were installed in Click and functionally validated with both stand–alone tests and with a network connection to a separate host running a conventional

IP implementation. The performance of the bindings was evaluated by comparing the number of CPU cycles and instructions required to process a packet in the unmodified and modified routers. Initial results are promising: the generated bindings for the fixed portion of the IP header are compiled into machine code essentially identical to the original C–coded bindings, thus incurring no performance penalty. The bindings generated for the more complex IP options syntax introduce a significant amount of overhead (for processing packets with options); however, the source of the overhead is evident. An objective stated in the future work plan is eliminating the overhead by modifying the Xyn code generator to produce C++ code with more obvious (to the back-end C++ compiler) pointer semantics and fewer branching constructs. In the worst case, a layer implementor will have to evaluate the tradeoff between the abstraction benefits of Xyn and its performance overhead, but even then, improved C++ compiler technology and increasing system hardware performance will broaden the applicability of Xyn.

1.3.3 Contributions

In presenting the theory, design, and implementation of a *syntax directed binding* facility, this dissertation advances the state–of–the–art through the following contributions:

- An improved programming practice that recognizes an abstraction boundary between the mechanisms of IO data syntax *navigation* and *access*, and the program semantics or policies associated with the IO data *values*; thus generalizing the well–known RPC/RMI paradigms to include applications such as video codecs and communications protocols that have traditionally utilized ad hoc bindings;

- The domain specific language Xyn and its lexicon Blex to succinctly specify the IO data syntax, and Xync to translate the specification into native access mechanisms for the syntax, ensuring consistency between specification and implementation;

- Xync codifies the implicit mapping between the (labeled) elements of a declarative Xyn/Blex specification and the (identically labeled) native programming language structures necessary to navigate and access the IO data elements;

- An inter–layer facility, MetaXyn, that exploits per–layer transfer syntax information to improve the type–safety of layer composition and to optimize IO data layout for minimal overall access overhead. This controlled breaking of the layer encapsulation is accomplished using the abstract specification of a layer; without exposing details of the layer's implementation.

- The inter–layer facility extends copy avoidance techniques traditionally constrained to the operating system environment into the presentation and application layers. The resulting improved discipline in the upper layer implementation is critical to exploiting the efforts put into optimizing the system layers.

In summary, this dissertation demonstrates how to achieve *effective data access in a software IO framework*; simplifying program development, improving type safety (both within a layer and in a composition of layers), and enhancing the reliability and performance of modern IO–centric applications.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes the architecture of the IO data access framework; both intra– and inter–layer. Chapter 3 describes Xyn, a meta–language for transfer syntax specification. Chapter 4 describes the implementation of IO data access bindings generated from a Xyn specification, and Chapter 5 analyzes the performance of the generated bindings. Chapter 6 describes the design of MetaXyn. Finally, Chapter 7 concludes the dissertation, and includes a discussion of future research directions. Appendix A presents a specification for IPv4 written in Xyn, which was used extensively in the validation of this work.

Chapter 2

Architecture

Chapter 1 described a model of IO processing systems organized around a vertically *layered* architecture, and consisting of several functionally orthogonal IO processing layers. A typical IO processing layer implements a service by exchanging information with one or more remote peer layers, passing the data through the intervening lower layers. Conceptually, the IO data path forms a *vertical cut* through the IO data processing layers [Hufnagel and Browne, 1989], along which each layer's operations consume an *import* syntax, update local state, and produce an *export* syntax. Communication with a peer layer conforms to a *transfer* syntax, while the layer presents its service using a *provider* or *user* syntax. Thus for an output operation, the user syntax is imported and transfer syntax is exported; on input, vice–versa.

The objective of this chapter is to describe and motivate the architecture of an IO data access facility that meets the challenges presented in Chapter 1. This architecture incorporates two sub–architectures: *intra–* and *inter*–layer. The intra–layer access architecture factors the IO data access mechanisms from each layer's semantic operations at the vertical cut, generalizing the per–layer IO data access interface by redefining it in terms of *binding* semantic operations to the elements of that layer's (abstract) IO data syntax. The intra–layer architecture is described in Section 2.1. The IO processing layers along the cut interact with each other through exchanged IO data. The *inter*–layer architecture captures the IO data access related (i.e., non–semantic) aspects of the interaction in a meta–layer protocol, exploiting shared information to optimize intra–layer performance and improve layer composition type–safety. Section 2.2 describes the inter–layer architecture. Section 2.3 gives an overview of the tools developed as part of this dissertation to facilitate deploying the binding architecture. Section 2.4 concludes the chapter with summary of the architecture's contributions.

2.1 Intra–layer IO Data Access Architecture

An IO data *syntax* is an abstract description of the structure of encoded IO data exchanged between (layer) peers. A syntax consists of a hierarchy of structural constructs such as aggregates, sequences, optional elements, a collection of value holding fields, and the rules mapping each abstract construct onto a bit–sequence. Chapters 3 and 4 discuss IO data syntaxes and syntax–to–binding transformations in detail; this section focuses on the requirements arising in accessing the IO data syntax abstractions through their concrete representation in a host programming language. At the endpoints of the communications link the IO data bit–sequence is stored in the host's physical memory. The *layout* of the bit–sequence describes the mapping of a syntax instance in a program's virtual–memory address space; i.e., the on–the–wire representation imposed on the host virtual address space.

Although syntax design is often influenced by the 8–bit–byte memory–access granularity pervasive in modern computers, in general, there is no assurance that the layout (i.e., "wire" representation) of a syntax structure will conform to the layout of corresponding host data structure dictated by the programming language implementation; or, even if it does, that the IO data will be delivered on a suitable alignment boundary. As described in Section 1.1, this arises from several sources: minimizing bandwidth consumption by packing multiple fields without regard to integral host type dimensions, the absence of inter–member structure padding to align integrally sized fields on integral boundaries, variable length (e.g., entropy) coding, or a mismatch in data representations between communicating hosts. Furthermore, a syntax construct might be split into segments by some lower layer in the transmission path, resulting in its discontiguous storage at its destination.

Since the IO processing operations cannot effectively directly access the IO data syntax, the syntax elements need to be presented in terms of the host programming language.

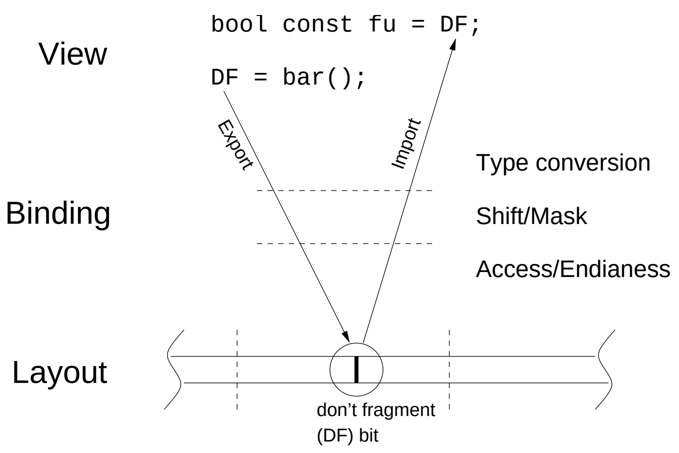

Figure 2.1: Binding a syntax field.

A *view* is a mapping of a syntax instance to structures and types in the host programming language. A view tailors a syntax to the expectations of semantic operations through the following:

- As an aid to navigating the syntax, it guides traversal of syntax structures;
- Presents each syntax field in a corresponding host type representation for access as a programming language variable;
- A view might present an adaptation, permutation, or subset of the syntax elements.

By allowing an IO processing layer to operate on syntax elements as if they had been defined directly in the layer's programming language, the implementation of the layer is simplified. In particular the layer implementor can focus on the semantics of the layer.

The objective of the intra–layer architecture is to facilitate the design and implementation of effective IO data syntax access mechanisms; that is, access mechanisms that transparently tie syntax views to layouts.

2.1.1 Bindings

An IO data *binding* exposes a syntax element embedded in the underlying wire representation *layout* to the layer's semantic operations through a *view.* A binding is an association between a host programming language construct (part of the layer's semantic implementation) and a syntax element embedded in the IO data layout. Bindings are transient: a binding is created when the IO data is imported into a layer and destroyed on export. The IO data fields correspond to host programming language value types, although interpretation might be necessary. A schematic of a syntax field binding is shown in Figure 2.1.

A binding mediates between wire representation (i.e., layout) and host representation (i.e., view) of the syntax element. In particular, the binding must accommodate differences in:

- Size (i.e., bits) and alignment (i.e., layout of a syntax element versus the host programming language type requirements;
- Byte order of multi-byte integer primitives (i.e., "endianness");
- Type encoding (e.g., sign–magnitude vs. twos–complement integers).

To further simplify IO data access, a binding is encapsulated with an access idiom, such as an assignment operator.

2.1.2 Navigation

In addition to bindings for accessing syntax fields, processing IO data requires bindings to *navigate* the syntax structures. Syntax structures have analogs in programming language structures. Common syntactic structures include *aggregates* (analogous to a C struct), *sequences* (analogous to a variable length array), and *choices* (analogous to a discriminated union). Much like the usability advantages motivating presenting syntax field bindings as programming language variables, traversing a syntax's structures can be greatly simplified by presenting the structure as its programming language analog.

Disparate requirements of syntax encoding for different applications gives rise to two IO data navigation styles: *sequential* or *random* access. Processing hierarchical, highly compressed, streaming protocols (e.g., MPEG) requires *sequential* access to syntax elements:

- At protocol granularity, top–level elements (e.g., sequence or picture header) contain parameters on which lower–level element decoding depends (e.g., picture size);

- At field granularity, variable–length fields enabling high compression require in–order decode;

- The encoding of an MPEG intra–coded–picture slice incorporates inter–, intra–, and skipped– (i.e., missing) coded macroblocks depending on compression achieved per macroblock; encoding or decoding the latter two require references to macroblocks in previous or subsequent frames.

Sequential traversal reduces the amount of redundant information required in an encoding, either by relying on the program to maintain previously decoded information in its state, or through special coding techniques such as entropy coding (e.g., Huffman coding). Since the encoding and decoding depends on the structure of the syntax, such transformations are called *syntax directed*.

In contrast to sequentially navigated syntaxes, processing lower–level (in the layered IO architecture) protocols requires *random* access to fields for fast (partial) protocol processing (e.g., IP routers):

- The syntax incorporates a fixed prefix designed to allow protocol header to be interpreted as a C–language struct;

- Access: pointer–to–buffer cast to pointer–to–C–struct; fields can be accessed with the pointer–to–member selector operator (`->`);

- Sub–integral sized fields (i.e., bit–fields) are packed into integral sized and aligned fields;

- Any individual field can be accessed directly from a handle (pointer) to the protocol header;

- Access overhead minimized by exploiting mature C compiler optimizers.

Random access navigation allows the sublayers of the protocol to operate independently of each other, thereby increasing the flexibility in implementing the protocol, and minimizes overhead by allowing each sublayer to directly access the fields on which it operates. This flexibility comes at the expense of increased size, but the tradeoff is acceptable since the amount of data is relatively small compared to, for example, a video stream. This type of "random" access is called *user–directed* protocol processing, since the environment provides bindings to the syntax elements, while any access and transformations are invoked by the user code (i.e., the protocol processing sublayers).

To accommodate this dichotomy of access style requirements, random or sequential navigation style is a per–syntax attribute, and a specialized binding implementation is provided for each style. A key architectural goal is to isolate differences due to navigation style to a minimal number of components in order to maximize reuse of other components. This is achieved through layering the navigation and field access mechanisms; note that this layering is orthogonal to the semantic layering of the IO data processing.

2.1.3 Layered Bindings

Figure 2.1 shows an example of a layered field binding. The lowest layer accesses the bit–sequence at the granularity and alignment allowed by the host architecture. Host–syntax endianness mismatches (integer byte ordering for multi–byte accesses) can be corrected at this layer by swapping the bytes delivered to, or received from, the next layer up. The shift/mask layer extracts the field's bits from the frames received from the access layer on an import operation, and merges (overwrites) the corresponding bits on export. The next layer translates between the syntax and host representations of the field; which ranges in complexity from a simple reinterpretation of a bit pattern as an unsigned integer, or translating between a bit pattern and a Huffman coding of an index into a table of zero–run–length/level values. Finally, the top layer presents an interface to the host encoding in terms of a programming language access idiom — typically an assignment operator taking the host type as an operand.

At the lowest level of the layered field binding, a sequential–access accessor requires a simple iterator over the underlying target data sequence, returning the requested number of bits, and then advancing that distance along the data sequence. The random–access accessor implementation is inherently more complex than the sequential access implementation, primarily due to the expectation of performance and access flexibility equivalent to C–language pointer and structure member access. Achieving this performance requires knowing at compile–time the displacement of the base address (i.e., the host address of the first byte in the current message structure) from some alignment boundary, and, for each syntax field, an offset of the field from the message base and the field length. Given these values, a good compile–time optimizer can reduce a runtime access to a syntax field to a pointer dereference and mask of unwanted bits (i.e., equivalent in cost to accessing a member of a structure defined in the host programming language).

This distinction between sequential and random access at the bit–sequence access level (and, of course, observing the use constraints implicit in the access style), allows the remainder of the field binding layers to be independent of the access style and thus reusable in either context. For example, the translation between a bit–field coded as a sign–magnitude integer into a programming language integer in twos–complement representation is identical for both access styles.

Higher order organizing constructs are also hierarchically structured. An *aggregate* is a collection of adjacent syntax fields. An aggregate's accessor consists of a sequence of corresponding member accessors and is presented as a structure through which the elements can be accessed by name. For example a syntax aggregate named A containing a field named b would provide access to b's value through the usual member selection operator $A.b$. A *choice* is a set of alternative aggregate branches and associated predicates to discriminate between the branches based upon the semantic (i.e., layer state) or syntactic (i.e., a prefix of the unconsumed bit sequence) context. A *sequence* is a collection of zero or more elements of some choice type, where one branch of the choice is empty — its selection indicates the end of the sequence.

Since a syntax intended for random access can still require variable length elements (e.g., a choice with branches of differing length, or a sequence), the layer programmer must supply runtime length information via an explicitly named "size" attribute in the prefix of a variable length structure's syntax specification. This dynamic (i.e., runtime) length information allows scanning the syntax at a level in the syntax hierarchy without incurring the overhead of fully examining the constructs of the levels below. To facilitate this access, the size attribute must be computed using no values beyond those in a prefix of fixed sized fields.

2.2 Inter–layer IO Data Access Architecture

Section 1.2 discussed syntax–related complications that arise when composing IO processing layers, and reviewed current approaches to remedy them and the limitations of these approaches. The dissertation proposes to remove these limitations through techniques to:

- Increase the composition's efficiency by initially delivering IO data at an optimal layout to minimize intra–layer copy overhead, and exploiting an unavoidable copy or copy–like operation to resolve layout conflicts;
- Integrate non–syntax–processing IO data layers, such as the user–system boundary crossing, into the inter–layer optimization mechanism to follow the "vertical cut" from application to physical IO interface;
- Validate the correctness of the layer composition through type–checking the inter–layer binding at the layer–layer interface.

These inter–layer techniques are based on revealing details about the intra–layer binding requirements, and then sharing this information via a meta–layer protocol in order to validate or enhance the composition. Since the information shared is derived from the *abstract* syntax descriptions of the IO data imported and exported by each layer, the sharing does not violate the information hiding that originally motivated the layering. Since IO data (by definition) crosses system virtual and physical boundaries, the system boundaries interposed into IO data path are incorporated in the inter–layer architecture, and must be accommodated by these optimization techniques. This section describes the architecture of an inter–layer binding optimizer.

2.2.1 Meta–layer Protocol

The intra–layer accessors described in the previous section impose *layout* constraints on the underlying IO data, requiring each IO data field to meet *alignment* and *contiguity* constraints in order to be accessible to the binding code. Since the intra–layer binding optimization depends on static parameterization, each of the layers in the collection of layers composed to provide a communications service for an application must receive IO data meeting its layout constraints or it will have to copy the IO data to a conforming layout. Then in order to minimize the overhead of copying, IO data should be delivered to the composed layers observing (to the extent possible) the intersection of the layout constraints. A combination of techniques are used to achieve this goal.

IO data contiguity is achieved through deployment of a meta–layer protocol called *application level framing* (ALF) [Clark and Tennenhouse, 1990]. By computing the intersection of minimum and maximum message sizes among the composed layers — this includes the message size of any networking layer(s) which is the *maximum transfer unit* (MTU) along the path between application endpoints — the source application is presented with a message size that is guaranteed to be passed intact through all layers to the destination application. This ensures each layer's field access bindings operate on contiguous IO data, without requiring copying of message segments to contiguous storage.

The per–layer alignment constraint is satisfied by an inter–layer alignment meta–layer protocol in two phases. For each layer, there is one or more alignments for which the layer could be statically optimized (i.e., compiled) that would minimize the number of CPU instructions to access the field for reading or writing. Furthermore, each layer accesses some portion of the IO data: typically either the entire message, such as a checksum to verify data integrity, or just a prefix, such as the IPv4 header. By considering this information across a proposed collection of layers — e.g., the TCP/IP stack — an algorithm built into the meta–layer protocol can compute an alignment that will minimize the overall access costs. This alignment value is used to parameterize the layer compilation, concluding the first phase.

The second phase repeats the protocol when a larger collection of layers is to be deployed as a complete communications service for an application. In this case, the algorithm attempts to minimize the cost of copying some or all of the IO data to meet each layer's static alignment requirements. The result of this phase is passed to the origin of the IO data (either application or IO device controller) to specify the alignment IO data is to be delivered to the layer composite.

An approach similar to that used for optimizing IO data layout is used to test for compatible layer types and to integrate non–syntactic elements into the IO data processing layer composite. The meta–layer protocols and algorithms for all of these are described in detail in Chapter 6.

2.2.2 Inter–layer Meta–protocols and Phases

A meta–layer protocol is used to exchange and aggregate the layer–specific details necessary to implement the inter–layer binding enhancements. Meta–layer protocols must execute at different times (layer configuration, deployment, and execution phases) to be effective:

- Copy cost minimization: configuration–time (e.g., graph) analysis; (generated binding) compiler can accordingly optimize bindings via a generated table of alignments.
- Provider and transport syntax type compatibility test: configuration–time evaluation report;
- ALF ADU/PDU path MTU discovery: bind–time protocol setup, dynamic adaptation.

Thus the meta–protocol must be available statically (at configuration time), at deployment or load time, and dynamically (during run time).

2.3 Tool Architecture

Xyn is a language in which to specify IO data syntaxes (see Chapter 3). Xync is the translator for Xyn: it facilitates creating robust IO data processing layers through the automated generation of IO data bindings. A Xync implementation schematic is shown in Figure 2.2.

- Xync: A translator taking a Xyn specification and producing a set of C++ bindings;
- The layer incorporates the C++ bindings to resolve layer (program) references to the syntax elements;
- When an IO data buffer is passed to a layer, the bindings provide the layer's semantic actions access to the buffer–resident syntax elements.

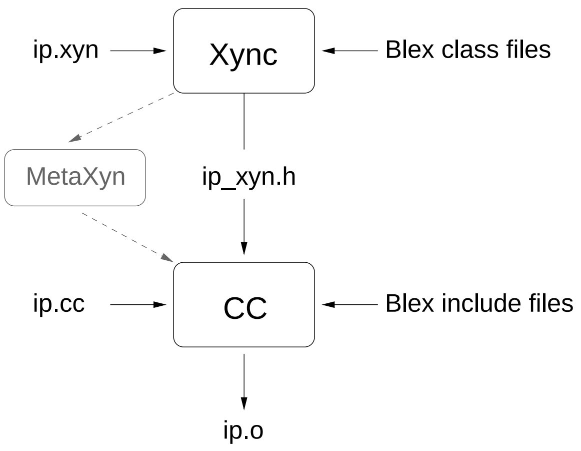

Figure 2.2: Xyn compiler (Xync): Flow through Xync translation system for the IP layer. IP protocol syntax specified in ip.xyn; IP semantic operations could be implemented across several modules; MetaXyn supplies feedback from composition of layers (protocol layer graph).

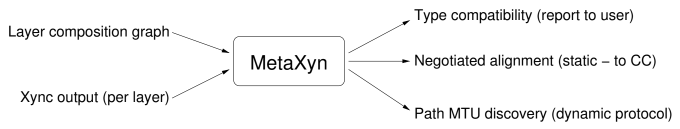
Layer composition graph
Xync output (per layer)
MetaXyn
Type compatibility (report to user)
Negotiated alignment (static – to CC)
Path MTU discovery (dynamic protocol)

Figure 2.3: MetaXyn inter–layer analysis and configuration protocol. Syntactic type compatibility report; Negotiated alignment; per–layer entry in C++ header file; Dataflow graph to route (local) dynamic path MTU discovery.

Blex is a bit–level lexicon that augments Xync with a specialized facility to express the terminals of the Xyn language (i.e., IO data fields).

- A collection of C++ template types mapping between syntax types and host programming language types;
- Supports compound (layered) types;
- Easily extended with new primitives by adding a template type and a Java class to the Xync translator.

MetaXyn is a meta–layer protocol to assist in creating optimal bindings between IO data processing layers whose transfer and provider syntaxes are specified using Xyn. An inter–layer implementation schematic is shown in Figure 2.3.

2.4 Contributions

The IO data access architecture presented in this chapter describes how to enable *effective IO data access* in the conventional vertically layered software IO processing architecture through distinct but coordinated *intra*– and *inter*–layer architectures:

- The *intra*–layer IO data access architecture defines a binding view interface to a layer's transient IO data: a view encapsulates the intricacies of binding IO data and presents the abstract syntax elements to the semantic operations of the layer in terms of conventional host programming language data access idioms;

- The *intra*–layer IO data access architecture further distinguishes between syntax structural constructs and their associated navigation requirements, and syntax fields and their associated access requirements; these are orthogonal problems and addressing them independently simplifies the complete IO data binding solution;

- The *inter*–layer IO data access architecture exploits intra–layer details to enhance intra–layer binding performance through copy minimization and end–to–end performance through application level framing (ALF), and to increase inter–layer binding type safety.

The architecture improves programming practice by introducing an abstraction boundary between the mechanisms of IO data access and IO data processing programs that operate on the IO data values. Chapters 3, 4, and 5 describe *Xyn*, an implementation of the intra–layer architecture and Chapter 6 describes *MetaXyn*, an implementation of the inter–layer architecture.

Chapter 3

The Xyn Language

In Chapter 2, the IO data access architecture describes how IO data bindings mediate between host programming language constructs (the layer or service implementation) and IO data syntax constructs (the layer or service's external representation). This chapter addresses the question: How should the architecture be instantiated?

On the one hand, low–level languages, in particular, C, provide bit–level manipulations necessary to operate on the syntax layout in host memory; however, the syntax design is obscured by the myriad details of the implementation. On the other hand, high–level languages, i.e., domain specific languages (DSL), allow expressing an implementation in terms of its design; however, the target domain (IO data binding) must closely match that of the language. Furthermore, in the case of IO data processing, efficiency is at least as important as succinct expression. A rhetorical question: Is C or an existing DSL appropriate for transfer syntax binding implementation? Or is a new "binding" language required? This chapter argues for a new DSL for IO data access. The chapter is organized as follows: Section 3.1 reviews existing approaches to implementing IO data access and their shortcomings, and introduces a new DSL called "Xyn" to implement the IO data access architecture. Sections 3.2 and 3.3 describe the Xyn language syntax and the bit–level lexicon "Blex", respectively. Section 3.4 describes the implementation of the Xyn compiler, called Xync. Section 3.5 concludes the chapter with an overview of Xyn's contributions.

3.1 IO Data Binding Languages

3.1.1 Conventional Approach to Transfer Syntax Binding

One approach to IO data binding is an ad hoc (i.e., per syntax) implementation of bindings using C and some preprocessor macros to manage architectural issues such as integer endianness, alignment, and contiguity. Internet networking protocols are typically implemented in C for the following reasons:

- Low–level system protocol requires low–overhead processing: access overhead must be minimized;
- Relatively simple navigation (one fixed aggregate per protocol, optionally followed by a variable suffix carrying message specific details);
- The syntax field layout intentionally avoids inducing alignment padding in a corresponding C language structure;
- The Internet protocols consist of many specialized, functionally complimentary but orthogonal services, which are incrementally implemented by different groups: hence, transfer syntax abstraction is not an overriding concern.

Specialized application syntaxes such as coded audio and video are also implemented in C (e.g., MPEG, JPEG).

- Performance sensitive, although access overhead can be overwhelmed by DSP costs (e.g., discrete cosine transform), performance concern tends to take priority over software engineering;
- Syntax utilizes packed–bit encodings that require complex bit–level manipulations, for which C is well–suited;
- Similar to networking protocols, a single organization does not implement enough of these to justify reuse concerns.

Translators from RPC/RMI interface definitions and ASN.1 data type definitions (i.e., DSLs) into transfer syntax bindings, typically emit bindings coded in C; again, because of C's affinity to low–level bit–manipulations, and the opportunity to exploit mature C compiler technology.

The Case for a Domain–Specific Language Based Approach

In contrast to the general–purpose programming language (i.e., C) implementations of IO data binding mechanisms, there are also several examples of language–based, or automated, translation between host programming language constructs and transfer syntax encodings:

- Transparently extending program composition techniques across process boundaries (including, especially, between hosts): RPC and RMI arguments are automatically encoded and exchanged between processes across a network;
- Specifying program data structures in an abstract language (e.g., ASN.1); then translating the specification into a set of routines to translate between a concrete instance of each data structure defined in a host programming language and a standard wire–representation of the abstract syntax element(s) corresponding to the data structure.

Existential evidence in the number of domain–specific languages in use for generating bindings indicates its utility. The benefits of an abstract specification and translator compared to an implementation using a general–purpose programming language include:

- Doubles as a formal specification;
- More closely reflects the intention of the syntax designer; simplifies future extensions and maintenance
- Enables design reuse in the development of new syntaxes;
- Translator embodies implementation reuse (and IO processing domain expertise);
- Facilitates analyses used to optimize layer compositions (inter–layer binding);
- Significantly more terse than an implementation of the bindings;
- Bindings are generated specialized to a host architecture (specification is not cluttered with these details).

Given these benefits of using a DSL–based approach to IO data binding when compared to hand–coded bindings, why are existing DSLs not applied directly to specifying (and generating) transfer syntaxes?

What About Current DSL Approaches?

Current DSLs facilitating IO data binding can be partitioned into two categories: 1) Those that provide IO data access in terms of conventional program data access by entirely isolating the facility's user (IO layer or application programmer) from the underlying details of the IO data binding; and 2) Those that reflect the structure of the underlying IO data, but are limited in scope, or for which IO data access is a means to another primary task, such as network packet classification for filtering. Each category has shortcomings when applied to the problem of describing general IO data syntaxes.

ASN.1 is a formal notation used for describing data transmitted by telecommunications protocols and belongs to category 1, above:

- Data structures are specified in ASN.1 using an abstract (i.e., programming language independent) structure description; intentionally avoiding any details of how the structures are to be represented on–the–wire (separation of concerns);

- Each encoding rule set (e.g., BER) is a (paper) specification of the *transfer syntax* associated with each ASN.1 construct; the transfer syntax implementation is embedded in the application–data to wire–encoding translator, and the translator implementor defines the application interface (i.e., access idioms);

- The recently proposed *Encoding Control Notation* (ECN), which provides for overriding of, and adding to, existing encoding rules, facilitates mapping ASN.1 to a wider range of transfer syntaxes, but spreads an application's syntax specification across three input modules (plus the implementation of the original encoding rule set);

- ASN.1's degree of abstraction (from its actual encoding) is intended to *isolate* implementors of IO processing programs (and applications) from the details of the associated transfer syntax; ECN is designed extend the applicability of ASN.1 to diverse transfer syntaxes, while continuing to maintain ASN.1's abstraction boundary, thus ASN.1 and its encoding facilities are inherently at odds with the requirement of straightforward specification of access to IO data;

RPC and RMI marshaling/serialization facilities translate between programming language constructs (e.g., C struct, Java class instance) and a transfer syntax (e.g., XDR, Java Object Serialization), and also belongs to the DSL category 1, above:

- Unlike ASN.1, these facilities generally assume a particular host programming language and wire–syntax (CORBA IDL/GIOP/IIOP is the exception as it is intended to be host programming language independent, although, in reality interoperability between vendors' implementations is not complete);

- Like ASN.1, RPC/RMI facilities operate on application data types and intentionally do not expose the transfer syntax representation.

DSLs belonging to category 2, above, include Universal Stub Compiler (USC) codecs, which translate between a packed (padding–free) sequence of fields and a corresponding C–language struct for a target host architecture:

- USC generates optimal translators for the fixed portion of protocol headers such as those found in the IP suite, for example, block–copying from source to target when layouts match;

- The specification language (IDL) expressly does not support variable length coded fields utilized in MPEG and other highly compressed transfer syntaxes;

- The specification language also does not support constructs other than an aggregation of fields (for example, the IPv4 options list requires sequence and a choice constructs); the documentation suggests that an additional layer should be implemented to translate between specifications using these constructs and USC.

Intel (NetBoost) Network Classification Language (NCL) and Agere Functional Programming Language (FPL) are also category 2 DSLs:

- Languages specialized for classification of network packets: access fields in the packet header in order to apply a predicate on the field values; dispatch the packet to the routine associated with the first valid predicate;

- Like USC, supports only the fixed, C–like aggregate of the networking header; similarly unsuited for specifying an access language for protocols in general.

In summary, current DSL approaches are either (intentionally) too removed from the transfer syntax, or operate on too small a subset of transfer syntax constructs in common use.

3.1.2 A New Approach to Intra–layer Binding

The intra–layer binding architecture describes how to structure an IO data access facility to present syntax structures in terms of host types and access idioms by hiding the intricacies of the bit–level accesses behind a *binding–view.* To realize this architecture, the intra–layer implementation exploits the hierarchical nature of transfer syntaxes:

- Protocol: message exchange sequence (might be bi–directional);
- Syntax: Per–message *structure*, including *choice*, *sequence*, and *aggregation* constructs;
- Lexicon: Syntax terminals; bit–level representation of value–types; mapped onto a bit–sequence according to the syntax instance.

The protocol is typically embedded in the layer semantics (e.g, remote procedure call) — its specification is beyond the scope of this work. *Xyn* is a language for specifying IO data syntaxes and is the key to automatic translation of a syntax specification into IO data bindings; Xyn constructs enable the IO program's *navigation* of a transfer syntax instance utilizing conventional programming language constructs. *Blex* provides the bit–level lexicon corresponding to the terminal set for syntaxes specified in the Xyn language; Blex accessors implement dynamic bindings to the syntax fields embedded in the IO data bit sequence, presenting the syntax fields to the semantic operations in terms of host programming language value types.

Much like conventional (programming) language processing, decomposition into syntactic and lexical aspects is critical to an effective solution, since the problems addressed are orthogonal:

- The Xyn specification explicitly expresses the structure of the syntax, to be navigated by the IO data processing;

```
syntax IPv4Net ;
...
public
IP
  : (  // 1. Protocol layer processing view (header only).
       v   : 0b0100          // IP version 4
       hl  : UIMSBF<4>       // [,5..15]
       hl_bytes : { (int) hl * 4 }
       ...
       len : UIMSBF<16>      // [,20..65535] total length in octets
       ...
       dst : Address         // destination address
       options : Options[hl_bytes - options.offset]
    |  // 2. Checksumming view (header only).
       chksum_header:(UIMSBF<16>)[hl * 2 ]
    )
    user_data:(Octet)[len - hl_bytes]
  ;
...
```

Figure 3.1: The top–level structure of an IPv4 Network header. IPv4Net: The syntax declaration; IP: the top–level message structure, syntax entry–point; A View block: One alternative is the aggregate of the labeled IP header fields, including the parameterized options component, the other a view to checksum the header; Followed by the untyped user–data (e.g., a TCP segment). Not shown are the definitions for the Option sub–message and its myriad constituent elements (see Appendix A).

- The Blex fields encapsulate the intricate bit–level manipulations required to access the syntax fields.

The Xyn compiler (Xync) takes the Xyn and Blex specifications and produces a collection of bindings expressed in the host programming language. C++ is currently supported, since it supports the C–style bit–level manipulations, operator overloading to present syntax types in terms of host types, and templates to efficiently implement Xyn's layered architecture.

3.2 Xyn Language

Xyn is a specialized (i.e., domain specific) language designed for concise specification of IO data syntaxes and to be amenable to translation into host programming language constructs that enable IO processing program navigation of a syntax instantiation. Xyn shares with general–purpose programming languages the ability to directly express the structures in the

```
Xynion    ::= label ':' Choice ';'              // Message container or component
Choice    ::= Aggregate ( '|' Aggregate )*      // Choice and View (alternative)
Aggregate ::= Element*                          // Empty alternatives are possible
Element   ::= label ':' ( EBNF | Ref | SemVar | Field )
EBNF      ::= '(' Choice ')' ( '?' | '*' )?     // SubXynion, optional, and sequence
Ref       ::= QualifiedID                       // Xynion reference (a label)
SemVar    ::= '{' ... '}'                       // User–defined computation
Field     ::= ...                               // Bit–level lexicon (terminals)
```

Figure 3.2: A synopsis of the Xyn grammar. Recursion is introduced by the *EBNF* production. One layer of productions comprises a lexical scope.

IO data syntax; however, since Xyn is intended to manage access to external data defined in an independent type system, Xyn focuses on the structure of the data, rather than operations on the data. Xyn shares with other IO oriented DSLs the goal of abstracting access to the IO data, but differs in that Xyn is designed to express the structure of the transfer syntax, rather than the structure of the application data. At the same time, Xyn maps syntax constructs directly to familiar host programming language constructs, facilitating IO program navigation of a syntax instance. Xyn, augmented with its translator, Xync, and bit–lexicon, Blex, implements the intra–layer access architecture of Section 2.1, addressing the intra–layer IO data binding challenges of Section 1.1. An example of a Xyn specification is given in Figure 3.1. The remainder of this Section introduces the Xyn Grammar, explains the processes of IO data parsing and generation in terms of Xyn, and describes the collection of syntax Types comprising Xyn.

3.2.1 Xyn Grammar

The Xyn language is for the specification transfer syntaxes. As such, it needs to accommodate constructs including messages, sequences, choices, views, and aggregates. A synopsis of the Xyn grammar is given in Figure 3.2. One or more *Xynion*s[1] tagged *public* are the entry points to the grammar and corresponds to a syntax message (unit of exchange). The body of a *Xynion* is a *Choice* block, or its degenerate form, a *View* block. A *Choice* block allows variations in the syntax: for example, the IPv4 header may be followed by one or

[1]*Xynion* is a play on *Xyn Union* — the *Choice* body results in a structure that behaves much like a discriminated union

more options; this is expressed as a sequence whose element is a *Choice* construct. The *Choice* construct consists of a branch for each possible option. Each branch of a *Choice* or *View* is an *Aggregate* of zero or more *Elements* (at least one branch would be non–empty). Each *Element* is labeled; the label names the *Element*'s accessor so it can be referenced from the IO data processing program. The *EBNF* or *SubXynion* construct introduces recursion to accommodate nesting of syntax structures, and closures (for sequences). *Xynion*s labeled *protected* provide for reuse of syntax expressions, simplifying and improving specification clarity (similar to subroutines). The *Reference Element* type inserts the named protected *Xynion*s into the syntax. These language elements express the structure of a syntax specified in Xyn; Chapter 4 describes their mapping to navigational constructs in the host programming language.

The remaining *Element*s are value types. Semantic variables are user computations necessary to describe some syntaxes (e.g., the IP header length is scaled by four to reduce its representation size). *Field*s represent the values encoded in the IO data; they are Blex constructs and are discussed in detail in Section 3.3.

The Xyn grammar was adapted from the ANTLR[2] [Parr and Quong, 1995] meta–grammar; i.e., an ANTLR grammar describes ANTLR itself. Various deletions, additions, and restructuring were required to tailor it to IO data syntax specifications. The Xyn (meta–)language is processed by ANTLR to produce a Xyn language parser, which, combined with a specialized Java library for Xyn AST processing and code generation, is called Xync (Xyn Compiler). Xync processes specifications written in the Xyn language to generate collections of IO processing program bindings for the transfer syntax elements described in the specification. The transformation of the Xyn specification into access bindings is discussed in detail in Section 3.4; the access binding implementations are described in Chapter 4.

3.2.2 Why EBNF?

Along with being specified using an EBNF grammar, the Xyn language is itself styled after EBNF. As discussed above, Xyn was derived from ANTLR, which utilizes an EBNF

[2]ANTLR: ANother Tool for Language Recognition.

style syntax and following this style was expeditious in getting a working prototype of Xync; however, Xyn's ultimate "style" is still under evaluation. Since C++ is the target language for Xyn's generated bindings (candidate targets such as Java or C share a similar syntax) and the writer of a Xyn specification is also likely to be the implementer of the associated IO data processing program, a C–like syntax for Xyn might make it more approachable. For example, [Object Management Group, 2002] provides a rich collection of types, including discriminated unions, and bounded and unbounded sequences using an annotated C–like syntax. While expressing Xyn using a C–like syntax could increase the visual correspondence between the specified syntax structures and their host programming language implementation, the current EBNF–style syntax has its advantages. First, since Xyn describes *syntaxes*, rather than application data types, EBNF is a natural representation. Second, Xyn is intended to double as an IO data syntax specification language, and presenting elements using a "label:type" ordering corresponds more closely to current transfer syntax specifications, such as MPEG [ISO/IEC 11172-2].

3.2.3 IO Data Parsing and Generation

Since the audience is likely to be familiar with programming language syntax analysis, several distinctions are drawn between that type of language processing and IO data syntax processing. Xyn–based IO data processing incorporates two functions not required of programming language parsing. First, the constructs generated from the Xyn specification must be able to generate as well as parse IO data. Second, unlike lexical analysis for language processing, which is typically implemented as a stand-alone token stream, the packed fields of an IO data bit sequence cannot be tokenized independently of the language parser and instead require syntax–directed lexical analysis.

Relative to parsing, IO data generation is straightforward, since the user (i.e., IO processing program) drives the transformations, inherently resolving ambiguities by selecting *Choice* branches and computing *Sequence* termination, without querying the associated Xyn constructs. Still, Xyn facilitates IO data generation by providing a syntax to assist in generating syntactically valid sentences, and encapsulating the low–level details of emitting the bit sequence.

```
protected IP.Options[ int length ]
  : list:( { SizeOctets() < length }=>
      option:( ... | ( Copy.false OClass.debmeas Option.TS )=> ts:TS | ... ) )*
    end:( Copy.false OClass.control Option.end )[ length - SizeOctets() ]
  ; // IP.Options
```

Figure 3.3: The IPv4 Options sequence utilizes semantic and syntactic predicates. The sequence must not have consumed *IP.hl_bytes* — passed via length parameter — octets so far (outer, semantic predicate); and, one of the Option branches' predicate must match (nested syntactic predicates); Otherwise, the sequence terminates, and parsing continues with zero or more *IP.Options.Option.end* elements to pad to a 4–octet boundary. The complete IPv4 syntax is given in Appendix A).

Lexical analysis of a conventional programming language transforms a character input sequence into a token stream, distinguishing tokens by a combination of interspersed "white–space" and special characters (e.g., braces or arithmetic symbols). In contrast, IO data is packed, hence providing no explicit lexical boundaries, and uses various ad hoc coding schemes to more compactly represent field values. As a result, tokenization of a packed bit–sequence requires knowledge of syntactic structure, which must be passed *from* the parser *to* the lexical analyzer, which can then extract and decode (conversely, encode and insert to produce) the lexeme.

Along with IO data generation requirements and syntax driven lexical analysis, a third fundamental distinction between programming language and IO data parsing is implementing lookahead, discussed next.

Disambiguation in IO Data Parsing

Disambiguation is one facet of parsing in which IO data parsing departs from conventional programming language parsing. Xyn's *Choice* and *Sequence* constructs allow structural variation among the instances of a Xyn specification. Selecting among a *Choice*'s branches and recognizing a *Sequence*'s termination require disambiguation at the variation points in order to parse (recognize) an instance for access to its fields. Note that this is only an issue for parsing, since generation is directed by the user.

Unlike programming languages, in which the language's syntax is designed to minimize ambiguities in its grammar, IO data syntaxes utilize explicit disambiguating information expressed directly in the syntax specification. Furthermore, the disambiguating information might be semantic (i.e., computed from program state) rather than syntactic (i.e., embedded in the unconsumed input). Hence, rather than utilizing static lookahead analysis to compute the lookahead sets for each grammar rule, Xyn's runtime selection of branches in a specification is performed with semantic and syntactic predicates.

- Syntactic predicate: Syntactic expression containing at least one literal
  (*0b01*)⇒ *Guarded expression*;
- Semantic predicate: Boolean valued expression
  {*a==1*}⇒ *Guarded expression*;
- Predicates are evaluated in the order presented in the specification;
- Syntactic and semantic predicates can be composed via (*SubXynion*) nesting;
- A syntactic predicate can always be replaced by a semantic predicate by parsing a prefix into local variables, then evaluating the semantic predicate over the variables; however, the syntactic predicate has the advantage of not consuming the prefix.

An example of predicates is the IP options list: a *Sequence* of a *Choice* construct, which is shown in Figure 3.3.

3.2.4 Xyn Types

A Xyn type corresponds to a syntax construct, which is presented as a type in the host programming language. Xyn types represent the navigational elements described in Section 2.1, such as choice and sequence constructs. Value–types, corresponding to syntax fields, belong to Blex, and are discussed below. The key concept underlying Xyn types is that each type is a collection of *references* to syntax elements. A constructed instance of a type holds a reference (a cursor) into the underlying bit–sequence, and incorporates the mechanisms to translate between the syntax encoding at the cursor and the corresponding host type, while navigating the syntax.

Xynions, SubXynions and References

A *Xynion* is a container holding one or more accessors to Xyn syntax elements. A *Xynion* tagged *public* corresponds to a protocol or application data unit (PDU or ADU, respectively) in the syntax's protocol. For example, the IPv4Net syntax public *Xynion* labeled IP in Figure 3.1 is the PDU for IP. A *Xynion* holds a reference to the underlying bit–sequence; each of the *Xynion*'s members is parameterized with its offset from the *Xynion*'s base. A *Xynion* tagged *public* is instantiated around the transient IO data content (or container) to be imported into (or exported from) the IO processing layer, facilitation binding the layer's operations to the *Xynion*'s member accessors. The *Xynion*'s members are also instantiated at this time.

A *Xynion* tagged *protected* is a component of other Xyn types; factored either for reuse or to improve the readability of the Xyn specification, and referenced from one or more points within the specification. Xyn allows forward references (i.e., no declaration is required), but not cycles — recursion is supported only at the grammar level. Protected *Xynion*s can also have parameters to receive semantic context passed from the reference's point of instantiation.

A *SubXynion* has the same body as a *Xynion* but is defined "inline" using the syntax:

label **:** **(** *Choice* **)**

Constructs nested in the *SubXynion* body are referenced through the *SubXynion* label; i.e., in ***a:(b:B c:C)***, ***a.c*** is an accessor for member type ***C***. Since each *SubXynion* is a unique instance (i.e., instantiated only in the context of its definition), the *SubXynion* body may refer to its semantic context at the point of instantiation; i.e., within its lexical scope. By having a *Choice* construct as its body, *SubXynion*s, (and the related closure constructs, *Sequence*s) introduce recursion to the Xyn grammar. Note that the recursion is at the abstraction level of Xyn grammar structures, not in syntax elements defined in a Xyn specification; the latter is not supported, since there is no means to terminate the recursive generation of a recursively specified type.

Choice and View

The *Choice* construct provides branching in a Xyn specification, allowing alternative types to appear in different instantiations of the specification. A branch selection is user–specified on export (via its label), while mandatory disambiguating syntactic or semantic predicates discriminate between branches on import. The predicate may be elided for an optional trailing default branch, which, if empty, makes the entire enclosing *Xynion* or *SubXynion* construct optional. The *Choice* syntax:

$$\textbf{Choice} ::= \boldsymbol{predA} \Rightarrow \ \boldsymbol{a}{:}\boldsymbol{A} \mid \boldsymbol{predB} \Rightarrow \ \boldsymbol{b}{:}\boldsymbol{B} \mid \ldots \mid \boldsymbol{z}{:}\boldsymbol{Z}$$

The branch body (after the ⇒) must be a labeled singleton element or a labeled *SubXynion* (or related closure); the label is used to reference the branch accessor. A *Choice* is a dynamic type, in that its actual type (branch) is not fixed until its predicates are evaluated on import or selected by the user on export. Thus its length can not be known until runtime, and thus any construct embedding a *Choice* itself becomes dynamic. An empty alternative renders the entire structure optional; a shorthand notation is provided for this construct:

$$(\boldsymbol{a}{:}\boldsymbol{A})? \equiv (\boldsymbol{a}{:}\boldsymbol{A} \mid \ )$$

An example of the *Choice* construct is the IP options list element type, which specifies that any one of many IP options can appear as an element of the options list (in practice, the number of options in the list is severely constrained by the list's maximum length of 40 octets).

A *View* is a degenerate (non–predicated) *Choice*, and provides alternative interpretations of a syntax component — unlike a *Choice*, all alternatives of a *View* are simultaneously valid interpretations of the same IO data. Selection of a *View* alternative is user directed on import as well as export. This requirement makes the *View* construct less practical for sequential access style navigation, since one of the features of that navigation style is driving the transformation with the syntax instantiation. The *View* syntax:

$$\textbf{View} ::= \boldsymbol{a}{:}\boldsymbol{A} \mid \boldsymbol{b}{:}\boldsymbol{B} \mid \ldots \mid \boldsymbol{z}{:}\boldsymbol{Z}$$

The attributes of a *View* (e.g., length) are based on the first alternative specified, and are used in computing attributes such as the length of the enclosing construct. Examples of *Views* are the field and checksum views of the IP header.

The *Choice* and *View* idioms are syntactically distinguishable by the former requiring a per–alternative predicate (with the exception of an optional trailing default alternative); while the latter has no such predicates. All members of a *Choice* or *View* share the lexical scope introduced by the enclosing *Xynion* or *SubXynion*, and hence must be uniquely labeled within that context.

Aggregate

An *Aggregate* is a collection of one or more Xyn *Element*s comprising the body of a *View* alternative; or promoted to the body of a *Xynion* or *SubXynion* in the case of *View* consisting of a single branch. An *Aggregate* can not be directly instantiated: it is a grammatical convenience. Hence, an *Aggregate*'s members are accessed through the label of the *Aggregate*'s enclosing *Xynion* or *SubXynion*. Within an *Aggregate* the offset of a field is computed as the offset of the previous field plus its length; for fields up to and including the first dynamically sized element (e.g., a member that is a sequence) this enables static computation of the binding parameters for each field accessor and results in highly optimizable binding code.

Sequence

A *Sequence* is a container for an ordered collection of homogeneous elements (however, the element type might be inherently heterogeneous — e.g., a multiple–branch *Choice* construct). A *Sequence* is a variation of *SubXynion* structure expressed using *EBNF* closures:

Sequence ::= (*ABC*)* | (*DEF*)+

Grammatically, the *Sequence* element is a *Choice* alternative block, and its predicated body supplies the termination computation. The specified *Sequence* element is implicitly appended with an empty branch (an existing empty default alternative is simply merged with the implicit empty branch), implying even a single–alternative body becomes a *Choice* construct. Parsing is greedy, so a non–empty default alternative is not allowed. Closures imply lazy (runtime) evaluation: On export, user–directed disambiguation indicates the user must terminate the *Sequence* (in this case, by advancing to the syntax element succeeding the *Sequence*); On import, semantic or syntactic context must be sufficient to recognize the end–of–sequence condition. When the *Sequence* is parsed for import, the *Choice* binding handle is queried each iteration and an accessor to the branch with the predicate evaluating true is initialized, and recorded or returned, depending on the implementation. When the *Choice* accessor evaluates to the empty branch, the *Sequence* is at its end. Along with the *Choice* construct, a *Sequence* is a *dynamic* syntax element, since its length can not be determined until runtime.

Xyn provides shorthand notation for common *Sequence* specializations:

- Counted *Sequence* (e.g., array with length specifier)
  Specialized syntax: ***(JKL)[N]***

- End–of–sequence Marker (to be consumed)
  Specialized syntax: ***(GHI)*[0b01]***

In addition to notational convenience, the implementation of the translation between syntax and host values of these constructs is optimized. The *Counted Sequence* requires a fixed size (at compile time) element type and a fixed length (at runtime initialization of the *Sequence* instance), and allows random access to the *Sequence* elements using C array notation (i.e. indexing operator). This is in contrast to the sequential access imposed by the general *Sequence* as a result of the variable length *Choice* element. The EoS Marker is tested for (lookahead) prior to evaluating the *Sequence* body. This behavior is desirable for a sequence of entropy coded values terminated by a reserved termination code, since the termination code is not mapped to a to–be–coded value belonging to the sequence (thus avoiding special case handling in a semantic predicate) and is likely to have a high probability of occurring (thus warranting the prioritized test).

Semantic Variable

Determining the structure of a syntax instance might depend on a runtime computation on the layer state or syntax field. For example, the IPv4 header length field is scaled by four from its actual value, which reduces the field's bit count (and implies all IPv4 headers are a multiple of 4 in length). The syntax of a *Semantic Variable*:

***label*:{ (*type*) *computation*}**

The computation is host programming language specific, i.e., it is not interpreted, although a language for computations that could be validated for syntactic correctness and translated into various host programming languages would be a worthwhile addition to Xyn.

The computation is constrained to refer only to *Element*s (including other *Semantic Variable*s) that appear ahead of it in the syntax. This is to ensure these bindings have already been constructed. This restriction is relevant primarily for sequential access; however, it is also important for random access when the semantic variable follows a dynamically constructed element (e.g., a *Sequence*).

Xyn also provides built–in *Semantic Variable*s (i.e., runtime attributes). For example *structureName*.Size() is the runtime length of a structure, dynamically computed for sequences and types with variable length components (e.g., one having a *Choice* element). Parameters passed into a *protected Xynion* are implemented identically to *Semantic Variable*s, the parameter is a named *Element* whose value is set to the argument value when the *Xynion* is initialized.

3.3 Blex: Bit–level Lexicon

Xyn's lexicon consists of a catalog of encodings for primitive types that appear in IO data syntaxes. The bit–level lexicon, called *Blex*, includes a variety of encodings, including bit–literals: *0b1001*; bit–sequence of length n, left bit first: *BSLBF*$< n >$; unsigned integer, most significant bit first: *UIMSBF*$< n >$; positive integer, most significant bit first: *PIMSBF*$< n >$ ... A Blex type is represented in Xyn by a *Field*; one of the Xyn grammar's terminal types. The interface between Xyn's structural types and Blex's value types minimizes coupling between their implementations: a Xyn construct (*Aggregate*) passes offset and alignment data to a nested Blex field, while the Blex field provides its size to its Xyn context. Isolating the details of providing bindings to segments of bit–sequences from the Xyn syntax structure and navigation simplifies both.

A Blex type implements the dynamic binding between a host programming language variable and a syntax element embedded in host memory for the purpose of providing read and write access to syntax terminals encoded in a bit–sequence. Unambiguous conversion between the bit sequence and host programming language type is a goal of Blex. For example, even though the conventional wire–encoding for unsigned and positive integer field types is identical, a distinction exists between them on the host, so the latter is represented on the host with the signed integer corresponding to the field value's magnitude. As discussed in Section 2.1.3 Blex bindings are layered to isolate the type adaptation issues from bit–field access, and simplifying implementation of each function. In spite of the layering Blex bindings are efficient. Reading or writing a Blex encapsulated variable has overhead similar to directly accessing a variable of the corresponding host type. Blex achieves this by implementing the bindings in the target host language, namely C++, and exploiting language features, such as C++ templates, to maximize static optimizations. In particular, in a random access environment, accessing a member of an Xyn *Aggregate* has the same overhead as accessing a member of a C *struct* through a pointer to the *struct*.

Blex provides a simple model for user–defined extensions (the same model is used for implementing the original catalog). For example, a sign–magnitude integer type can be added to the catalog following these steps: 1) writing a C++ translator (type adapter) operation that converts between the bit–sequence sign–magnitude representation and the host's twos–complement representation by multiplying the magnitude by the sign–extended sign bit; 2) wrapping the type adapter around a Blex built–in bit–sequence accessor; and 3) adding an identically named Java class extending *Xyn.<BlexTypeName>.java* in Xync's classpath to cause Xync to emit the type instantiation when it encounters a field declared to be of that type.

Blex implements a variety of more complex encodings, including an Enumeration type using the syntax:

$$\boldsymbol{label} < \#bits, enum >$$

that is defined in the style of Xyn's *Choice*, and maps a set of bit–literals each of $\#bits$ length to a host programming language enumeration type (e.g., a C++ enum). In some cases, a significant amount of bandwidth can be saved by transmitting an index to a table that is shared by the endpoints (either fixed or transmitted beforehand). Coded types are a generalization of enumeration, in which the mapping is from a bit–literal code to an arbitrary host programming language type. The coded type tag corresponds to an index into a table holding the host value. The coded type is further generalized by allowing both fixed–length (as is the enumeration) and variable length codes. An example of the latter is the entropy (Huffman) coding of (zero–run–length,level) pairs representing coefficients of an image, used pervasively in the coding of digital images and video for trasmission.

3.4 Xync

Xync, the Xyn Compiler, transforms a Xyn specification into a collection of syntax navigation and access bindings implemented in C++. Xync consists of four components:

- Xyn parser. An ANTLR [Parr and Quong, 1995] generated (meta)parser, reads the Xyn specification and builds an AST consisting of nodes corresponding to the Xyn Types described in Section 3.2.4. The AST consists of a list of *Xynion*s defined at the syntax level, and each *Xynion* is the root of a subtree of its elements;
- Xyn AST transform. The ANTLR implemented AST is transformed into a specialized (i.e., hand coded in Java) AST[3];
- AST manipulation. The AST transformation is completed, which includes walking the AST to bind type references to *Xynion* definitions, and optimizations such as merging adjacent literal fields in syntactic predicates;
- Code generation. The final series of AST walks is to emit the C++ header file containing the binding definitions. This includes steps to emit declarations for a class corresponding to each *Xynion*, then a definition of each class, including definitions for each member. For example, *SubXynion*s are implemented as member classes;
- Blex. Each Xyn *Field* definition is an instantiation of templates defined in the Blex catalog. Each definition is composed from more than one class as a result of Blex's layered implementation.

Xync's generated header file is then included in the IO data processing program. Each of a layer's message type(s) (i.e., *public Xynion*s is accompanied by a *MakeAccessor* function to

[3]Ideally, the AST manipulation would have been implemented entirely within the ANTLR AST framework; however, the stand–alone component was necessary since the ANTLR release available to implement Xync does not support heterogeneous AST nodes needed to encode the Xyn–type specific information needed for processing the AST — only text strings.

be invoked by the IO data processing program to construct the message type's bindings to the passed–in IO data bit–sequence. Subsequently, the IO data processing operations bind their variables directly to the IO data accessors. An actual implementation is described in Chapter 4.

3.5 Xyn and Blex Contributions

Section 1.1.1 presented a list of IO data binding challenges arising in the transformation between programming language data types and an IO data transfer syntax. This section presents the list of (abbreviated) challenges and Xyn's response, followed by a overview of Xyn's contributions.

- *IO data is transient and external to the processing program (i.e., service).*
  ⇒ Xyn's binding–views transparently provide semantic operations with import and export access (e.g., field assignment) to the transient IO data;

- *A mismatch between programming language type representation at application endpoints, and between an application endpoint and the external typing of IO data "on the wire."*
  ⇒ Xync/Blex generated bindings transparently adapt between host and on–the–wire type representations;

- *Complex and diverse IO data syntaxes and element types.*
  ⇒ Xyn language design follows from existing formal specifications of transfer–syntaxes — essentially an attributed formal syntax specification that allows direct translation from a syntax specification into IO data bindings; Blex's catalog can be extended with new syntax field types using a layered approach that simplifies implementation and simplifies reuse; The layered implementations of Xyn and Blex isolates Xyn syntax specifications and Blex field type mappings from the particular navigation access style (i.e., random or sequential) required by a transfer syntax's processing, increasing reuse.

- *Mismatch between programming language defined data access mechanisms and IO data syntax access requirements.*

⇒ Xync/Blex generated bindings perform the intricate bit–level operations required to access syntax elements;

- *Mismatch between IO data syntax structures and programming language data organizational structures.*
  ⇒ Xyn notation includes a *Choice* construct to handle branching in a syntax and corresponding to a discriminated union, and *Sequence* construct with notations for common end–of–sequence indicators;

Code to perform the dynamic binding of IO data must be produced when a program (layer) is implemented or when the IO data syntax is extended; by addressing the list of intra–layer binding challenges, Xyn relieves the layer implementer of the low–level data manipulation and parsing techniques necessary to process the IO data.

In summary, this chapter describes how to factor data access from IO program semantic processing, and the benefits of this increased abstraction:

- Xyn/Blex implements the *intra*–layer architecture and relieves its users of much of the tedious and error–prone coding required to implement IO data access mechanisms;
- Identifying and isolating the syntax navigation constructs of Xyn and the syntax field access mechanisms of Blex simplifies implementing each;
- Xyn/Blex allow the IO processing layer implementer to work with an *abstract* interface to the IO data syntax; the abstract Xyn specification is translated into concrete bindings;
- Xync/Blex generated bindings transparently mediate between host programming language access idioms and intricate bit–level operations need to manipulate the elements of an IO data syntax instantiation;

In demonstrating the feasibility of implementing the intra–layer architecture described in Chapter 2, this chapter improves programming practice by successfully introducing an abstraction boundary between the mechanisms of IO data access and IO data processing programs that operate on the IO data values. Chapter 4 describes an application of Xyn.

Chapter 4

Implementation

Programmers take the convenience of programming language data accessors for granted; for example, in C++, program data structures are first–class objects. The language provides simple notations to access data fields, and the language implementation transforms the notation into efficient execution code. Access to IO data is more complex. C++ is designed for a model hardware architecture that is integer and byte oriented. While C++ incorporates a bit–field construct for sub–byte or non–integer sized field aggregates, in general, even C++ bit–fields cannot directly map IO data syntax to native host data structures. C++ field ordering and padding are implementation dependent, while variable length field encodings are outside the model architecture. Furthermore, higher–level organizational constructs, such as the C++ library's *vector* manage their own memory (including copying new elements into the internally allocated storage), which conflicts with fundamental requirement to provide bindings to data allocated external to the program.

The utility of the Xyn binding facility results from its providing transparent access to the abstract syntax elements embedded in the transient IO data. A Xyn binding implements a *view* of syntax content (on import) or container (on export), expressed through familiar C++ types (i.e., interfaces). Hence, layer semantic operations can bind to the abstract IO data syntax elements in much the same way as they bind to native programming language defined data types.

The previous two chapters described an IO data binding architecture and a language for expressing the syntax of IO data. The Intra–layer architecture of Section 2.1.3, described how IO data binding functionality can be factored into a coherent collection of (functionally) orthogonal layers. Section 3.2 described Xyn, a language that captures organizational aspects of transfer syntax, and its mapping to the architecture elements, while Section 3.3 described Blex, a bit–level lexicon for Xyn's value types embedded in the IO data syntax. This chapter presents the realization of the architecture in the layered binding code generated from a Xyn specification, specifically, the strategy and techniques behind the C++ implementation of Xyn. Section 4.1 describes the layered implementation architecture. Sections 4.2 and 4.3 detail the Blex field and Xyn navigation construct components of the implementation and their composition. Section 4.4 concludes the Chapter with a summary of the implementation's key attributes and contributions.

Note that in this and subsequent chapters, "RA" stands for "random access navigation style," while "SEQ" stands for "sequential access navigation style." While contributing to acronym overuse is regrettable, these terms are used so frequently in the sequel, the tradeoff seemed worthwhile.

4.1 Binding Implementation Architecture

The previous chapters describe several challenges arising in IO data binding, and these are reflected in, and hence must ultimately be resolved in this implementation. Section 2.1.3 described the factoring of binding functionality into orthogonal layers, and Sections 3.2 and 3.3 describe a mapping between the elements of a syntax specification language (Xyn) and lexicon (Blex), and the architectural layers. The *implementation* architecture consists of the guidelines for realizing each Xyn and Blex language element (and constituent layers) in C++ code, and how the resulting implementation components are instantiated to form a cohesive binding facility for a specific Xyn syntax.

The value of the decomposition or factoring into layers is premised on discovering clusters of tightly coupled functionality, then presenting each collection of functionality in terms of an interface. Layering constrains the decomposition to an acyclic dependency graph, simplifying implementation and composition. Furthermore, various implementations of each layer's interface can provide different behavior, satisfying a wider range of clients and potentially combinatorially increasing reuse. In this C++ binding implementation, the layering observes the functional decomposition arrived at in the previous chapters; however, the implementation's interfaces are influenced by both the layer function and the C++ language's capabilities and constraints, in particular, those that must be met to enable automatic performance optimizations.

Each Xyn and Blex syntax element (layer) implements a set of interfaces, where each interface corresponds to a phase in the deployment and use of that syntax element. The set consists of *generation*, *composition*, *instantiation*, and *access* interfaces, which are described next.

The Xync code generator is organized according to the Xyn grammar: a *generation* interface is part of each grammar element abstraction in Xync, and variations within a grammar element (i.e., in the behavior of an actual syntax element) are captured where possible by generator interface parameters, otherwise, distinct code–generator implementations are invoked as required. Each generator consists of a boilerplate code emitter or C++ template, which Xync specializes with the labels and other parameter values corresponding to each syntax element of a Xyn specification.

A static, i.e., compile–time, *composition* interface provides the parameters required by C++ static typing (e.g., template parameters), along with constant values that facilitate performance optimization. The nesting and aggregation structure of this composition follows exactly from the Xyn specification of the syntax, and enables automated composition (aggregation and nesting) of the syntax element implementations. Parameters include navigation style, field width, and, for RA[1], expected alignment of the element.

An *instantiation* interface serves to initialize an accessor with a syntax instantiation (e.g., protocol message), requiring accommodating the transient nature of IO data from a layer's perspective: import $\rightarrow$ transform $\rightarrow$ export. Binding transient IO data requires ensuring the alignment and size requirements of a compiled element accessor are met; this can require copying or continuation support from the implementation. Finally, the *access* interface provides appropriate programming language data access idioms to the syntax elements. The consistent use of these interfaces throughout the Xyn implementation simplifies binding–code generation and facilitates its ultimate use in layer semantic operations.

[1]Recall that RA stands for "random access navigation style."

The fundamental decomposition dimension of the Xyn architecture is between navigation and value constructs in the syntax, which is reflected in separate implementations for Xyn (navigation) and Blex (value) constructs. A Blex type implementation is a bidirectional mapping between a syntax element, such as a sign–magnitude integer, and an accessor for a corresponding host programming language value type, such as twos complement integer assignment. Blex types implement the Xyn *Field* type, which is Xyn's interface to its lexicon and also provides the layer's IO data processing program access to the instantiated syntax fields. The Blex implementation is described in Section 4.2.

A Xyn navigation construct implementation maps between a syntax organizational construct (e.g., message, choice, or sequence) and an accessor for a corresponding host programming language structural type (e.g., an aggregate, discriminated union, or array). The Xyn implementation is described in Section 4.3. The result of Xync processing a Xyn specification is a collection of logical accessors produced from the composition of implementation layers; matching each abstract syntax element to a C++ type.

4.2 Implementing Blex Fields

Blex types are an implementation of Xyn grammar's *Field* abstraction. A Blex type implements an access binding to a syntax terminal element encoded in the underlying IO data, and presents it in terms of a host programming language value type. The key challenges the Blex implementation must satisfy:

- The external representation of the IO data syntax requires bit–level access on byte and integer access oriented systems;
- Due to the IO–centric nature of the target applications, access efficiency must approach that of access to native host programming language constructs (which is assumed optimal);
- IO data syntax encodings utilize a rich set of techniques to minimize entropy in the IO data stream. These include algebraic value transformations and mapped (table) encodings, and both static and dynamic elements;

- Accuracy and portability require exposing sufficient attributes to ensure accurate translation between the embedded bit–sequence and the corresponding host type on a variety of host architectures; explicit parameterization to assist portability.

The Blex implementation addresses these challenges through an implementation observing the architectural layers described in Sections 2.1.3 and 3.3. The implementation layers, described next, are (ordered from raw IO data to the host programming language binding interface): buffering and bit–access, translation and interpretation; and access idiom (user interface).

4.2.1 Bit–access Layer

The bit access layer provides read and write access to a requested segment of bits within the IO data bit–sequence, and interacts with its user using an unsigned integer value (i.e., a register holding the segment's bit pattern in its least–significant bits, padded with zeros for unused bits). This layer helps to isolate RA or SEQ[2], expected alignment of the element. differences from subsequent layers by providing a an implementation tailored to the navigation style. The layer presents a *value* interface to its client Blex translation layer, that is shared by RA and SEQ implementations; however, the layer presents RA and SEQ specific navigation interfaces. The RA implementation provides a frame–offset navigation interface, in which a sequence of frames matching the host's natural access boundary (e.g., integer) overlays the in–memory bit–sequence, and an access starts at an offset from a frame boundary. RA requires fixed length bit–fields in order to statically compute the offset of every field within an aggregate prefix (i.e., the enclosing Xyn navigation construct). The field's offset then allows static computation of the field's resident frame and offset within that frame. This technique is essentially identical to C++ bit–field access, except the Blex implementation allows a field to fall on frame (integral) boundary by splitting the request on the boundary, which better accommodates the packed nature of IO data. Since the offset computation depends only on static values, a C++ compiler can largely optimize away the computation. Thus the amortized access overhead is comparable to accessing a C++ bit–field.

[2]Recall that RA stands for "random access navigation style" and SEQ for "sequential access navigation style."

In contrast to the RA frame–offset interface, the SEQ implementation provides a *setbits–getbits* style syntax navigation interface. On export, *setbits* writes a value as a bit–segment (sequence of specified length) at the sequence cursor, then advances the cursor to the first bit beyond the segment. On import, getbits reads a bit–segment, then requires a separate call to advance the iterator. For a fixed–length field, the distance to advance is the syntax–specified size of the field. For a variable length field, a typical decoding strategy is to retrieve a segment the length of the maximum sized member of the VL set, interpret it (including discovering the actual bit–length), and advance the cursor the field's length. By default, the access layer returns 0–valued bits in case of underflow at the end of an underlying buffer segment. Along with supporting variable length fields, the distinct advance operation can also be used to update a field (assuming no change in size), and to skip uninteresting fields, providing the distance to the next desired field is known.

The access layer also accommodates an "endianness" mismatch between the syntax and the host. This problem arises when a syntax encodes its fields using one data ordering, while the host interprets the values in memory using a different ordering. In particular, a syntax might encode multi-byte integers from most–significant–bit to least–significant–bit in the bit–sequence, resulting in the most–significant *byte* of a syntax integer encoding residing at the lower addressed end of its field in memory (called "big–endian"), while the host architecture specifies that the least–significant *byte* of an integer appear at the lower addressed end of its storage (called "little–endian"). The C++ bit–field member ordering also varies between host architectures. The original Internet Protocols and bit–oriented syntaxes such as MPEG are big–endian, while protocols developed originally for systems running on the Intel x86–based PC architecture are little–endian (reflecting that characteristic of the x86 processor). To accommodate a host–syntax endian mismatch, the bit–access layer must reverse the byte ordering of any access falling on a byte boundary. This technique works for arbitrary, but sub–integer, sized segments, not just integrally sized and aligned fields. A SEQ implementation can obviate endian–mismatch concerns by re–filling the access buffer one byte at a time, with a possible performance penalty.

4.2.2 Translation or Interpretation or Encoding Layer

The Blex implementation layer built on the bit–access layer translates between the bit–segment (encoded) value held in a syntax field and provided by the underlying layer, and the host representation of that field. The layer implementation consists of a translation function whose domain is the encoding's bit–patterns and whose range is that of the host value type is required (for import), along with its inverse (for export). The implementation accommodates both those Blex types that can use a algebraic (functional) mapping and those mapping an index to a table entry. The former can be implemented by specifying a function parameterized with the syntax field size (usually related to the range of the encoded type). At one extreme of this encoding are the unsigned integers and bit–literals that exactly match their corresponding host encodings (including width). More sophisticated encodings include the transformation between a syntax sign–magnitude integer representation (actually a composition of two bit–segments) and a host's twos complement signed integer representation. Blex's bit–literal types are implemented as a C++ type *bool* conversion: the host value is *true* when the IO data field matches the specified syntax value, and *false*, otherwise.

For table–based encodings, the current Blex implementation can generate bindings for only simple enumerated types (collections of bit–literals) from a specification, while more complex table encodings must be provided to Xync as a translation function and its inverse. The generated enumerated type transformation is implemented as an *switch* statement embedded in a function[3], while the translation function for the MPEG standard's Huffman encoding of zero–run–length, level pairs was implemented by hand. A future Blex implementation could include a table specification language and a translation function generator.

[3]A direct mapping between the bit–segment and the enumerated type is valid for the C++ *enum* type and is used on export; however, the embedded switch statement provides error checking on import. An optimization would be to elide the switch statement when the enumerated type covers all possible values of its field.

4.2.3 Host (User) Interface Layer

Since Blex types are value types, the typical host–interface access idiom is the assignment operator. The implementation of this layer consists of a C++ assignment operator override for export and a type conversion operator for import. A function object incorporates these members as interfaces to the type conversion functions from the translation layer. Since parameters are passed by reference, and results are either a return value constructed in–place (algebraic conversions and inlined table entries) or a returned reference (explicit, static table entries), implementation overheads are negligible.

4.2.4 Layer Composite

While the purpose of a Blex type is to map between an abstract syntax value represented as a host type and a segment of the IO data bit–sequence, the intention of Blex's layered implementation is to facilitate reuse along the various dimensions of Xyn's implementation. For example, the bit–access layer provides a an RA or SEQ navigation–style specific interface to the Xyn *Aggregate* (the Xyn layer that interacts with Blex), while providing a common bit–segment interface to the Blex translation layer. In turn the translation layer builds on the bit–segment (i.e., unsigned integer) interface and provides a function interface to the user–interface layer, which, then provides a host value–type interface to the encompassing IO processing layer's semantic operations. The layering facilitates flexible and terse Blex type expressions, enhancing reuse by composing a variety of instantiated and generated translation layers with the instantiated bit–access and user–interface layers.

The Blex layer implementations make extensive use of C++ templates, operator overloads and inlining. A typical Blex type instantiation includes the host access–idiom, such as assignment, parameterized by the conversion name, and bit count and host type, or table name, and bit–access style. Generally, for a RA segment (i.e., a Xyn *Aggregate*) of fixed length fields, template and inlined transformations allow the compiler optimizer to reduce access to comparable operations on host types (e.g., the overhead of shifts and masks to extract or insert packed operands is comparable to C++ bit–field access). While efficient machine code is critical, the complexity arising from extensive use of templates is largely hidden from the layer implementer. The template instantiations are emitted by the

Xync only after the Xyn specification has been validated to be syntactically correct, and are largely boilerplate code wrapped by additional Xync output providing parameterization.

In addition to reliability and performance, a key benefit of the Blex catalog is that Blex types are shared among protocol family members, such as MPEG and H.263 video codecs, or within the IP suite of protocols. As described in Section 3.3, the Blex catalog is designed to be extended by a syntax implementer.

4.3 Implementing Xyn Navigation Constructs

The Xyn implementation maps IO data syntax structures to host programming language structures, enabling access to the IO data through conventional programming language data access idioms. The Xyn navigation accessors traverse abstract syntax structures imposed on the IO data, interacting with the underlying bit–sequence through the Blex elements instantiating Xyn *Field*s. An implementation of the syntax navigation architecture must accommodate several challenges, including:

- The transient nature of IO data: accessors must be constructed for each message imported or exported;
- Buffering the IO data, including dynamically coping with a mismatch of alignment or capacity;
- Accessing static constructs with efficiency approaching the corresponding access to a host language structure field;
- Dynamic *Choice* and *Sequence* constructs: Variable length (element count) *Sequence*s and predicated/branching *Choice*s result in a syntax structure that varies among a syntax's instantiations;
- Factoring RA and SEQ navigation style differences to maximize reuse;
- Creating navigation accessors that mimic host programming language structural types, so they can be used naturally in the layer's semantic operations.

The Xyn implementation addresses these challenges using a combination of static (compile time) and dynamic (runtime) techniques.

The implementation exploits the structure inherent in the Xyn grammar. Reflecting the grammatical layering, the series of productions

```
(Sub)Xynion ::= Choice|View ::= Aggregate* ::= Element* ...
```

comprise a single level of recursion in the Xyn grammar, and the layer composition represents a single lexical scope in the implementation. Each grammar rule corresponds to an interface, thus each of the rule's alternatives provides implementation to match its particular IO data syntax processing requirements. The uniformity in interfaces facilitates composition (both aggregate and nested) of various Xyn features into a particular Xyn specification's instantiation. The following is a bottom–up description of the Xyn navigation construct implementations, starting with Xyn's terminals — from the Element production.

4.3.1 Fields and Semantic Variables

*Field*s and *Semantic Variable*s are Xyn's value types; they are abstractions of data values, while other Xyn types are abstractions of the IO data organization. A *Field* is a place-holder in the grammar for a labeled Blex expression — described above in Section 4.2 — and Xync instantiates the definition appropriately parameterized for RA or SEQ. Xyn's other value type is a *Semantic Variable*, which does not appear in the IO data, but represents a computation to aid in syntax navigation. For example, the IPv4 *ip_hl* field's value must be multiplied by four to reveal the actual header length in bytes. In the current implementation, a *Semantic Variable* is useful for import only. A *Semantic Variable* is implemented using the C++ function object idiom; in this case, holding a function generated from the *Semantic Variable* entry in the specification and initialized with a reference to the enclosing lexical scope (i.e., the object of which the *Semantic Variable* is a member). The Xync *Semantic Variable* implementation is also used for *Xynion* parameter passing and *SubXynion* lexical scope handles.

4.3.2 Aggregates

A Xyn *Aggregate* consists of a contiguous collection of *Element*s comprising a Choice branch or View alternative; it is Xyn's interface to Blex's Field bindings. The *Aggregate* construct essentially corresponds to a C++ "plain old data" construct (i.e., a C *struct*), with any field alignment padding elided; however, an *Aggregate*'s *Element* accessors are only available through the label of the *Xynion* or *SubXynion* introducing the lexical scope containing the *Aggregate*. An *Aggregate* is the fundamental Xyn binding unit: in particular, selecting an *Aggregate* in the Xyn parse or generation implies all the *Aggregate*'s *Element*s must be (on import) or will be (on export) present in the contiguous segment at the current point in the syntax instantiation. The distinctions between RA and SEQ navigation constructs are largely isolated to the *Aggregate* implementations (similar to lowest level of Blex bindings); these implementations are discussed next.

An *Aggregate* generated by Xync for a SEQ syntax provides for a sequential (linear) traversal over member *Element*s, and advances the IO data cursor with each field processed. This implies only the accessor of the *Element* at the cursor is valid, and the syntax implementer is responsible to retain any value required subsequent to the cursor's advance.[4] When control returns from the function processing the *Aggregate* the IO data cursor must be positioned at the start of the next Xyn construct in the syntax instantiation. The IO data cursor is passed by reference, so an advance in the function processing an *Aggregate* appears in the caller's context when control returns.[5]

An *Aggregate* generated by Xync for a RA syntax consists of a prefix of static fields, optionally followed by a suffix of dynamic fields. This organization is designed to provide access to the static prefix fields (and, perhaps the first dynamic field) at a cost similar to C++ structure field or bit–field access. The static prefix may include fixed–size *Field*s and *SubXynion*s, and references to fixed–size *Xynion*s. The *Aggregate* is statically parameterized with the base alignment from its enclosing *Xynion* or *SubXynion*. The first *Element* of the static prefix takes its offset from the *Aggregate* base. For each succeeding *Element* in the static prefix and the first dynamic *Element*, the *Aggregate* instantiation statically computes the offset for each *Element* binding as the sum of the previous *Element*'s offset and size (Note that *Semantic Variable*s do not appear in the IO data and have a size of zero). Hence:

[4]Enhancements to Xyn to remove this burden from the layer implementer include implementing field–value or cursor–position caching bindings, or implementing an extension to the the action language suggested in Section 3.2.4 to automatically retain a value for any *Field* mentioned in a subsequent *Semantic Variable*.

[5]Alternatively an index could be recorded, and IO data length of the Aggregate returned. In some ways, this might be more explicit and hence less confusing to a layer implementer.

- The *Element* offset parameterizes its *Field*'s Blex type IO data binding, *SubXynion*, or *Xynion* reference (which depends on the *Element*'s instantiation);
- Each *Element* accessor (binding) is compiled assuming the specified alignment, facilitating optimal access through the underlying frame binding;
- If arriving IO data is misaligned, the buffer layer underlying the *Aggregate* must accommodate, typically copying the IO data to a properly aligned buffer.

Subsequent dynamic elements are statically parameterized with the alignment computed for the first dynamic element; the underlying buffer layer must accommodate any runtime IO data alignment mismatch. All dynamic elements must be accessed in their order of appearance in the Xyn specification so that the actual size of preceding elements is known.

Xync generates a RA *Aggregate* by emitting each *Element* in the static prefix as a member of the enclosing *Xynion* or *SubXynion* class, along with static size and offset value expressions, and initializes the *Element* with a reference to the *Aggregate*'s IO data buffer (which is ensured to match the *Element*'s alignment requirements). Dynamic members are initialized in order of appearance and to the extent necessary so that an element's attributes are available to initialize any succeeding dynamic element. On export, dynamic *Element*s are initialized with a size of zero. When the layer program's processing of the syntax's instantiation reaches the *Element*, it is (re)initialized to the type requested by the program.

4.3.3 Views

A Xyn *View* is a collection of one or more *Aggregate*s. Analogous to a C++ *union*, each *Aggregate* provides an alternative interpretation of the same IO data segment. One typical use of a *View* is an alternative consisting of the syntax definition, while another alternative presents the IO data as an octet sequence; i.e., "raw data," used for transmission error checking. Such a multiple alternative *View* is typically an RA construct. For SEQ, multiple branches can be specified but only one alternative can be traversed without resetting the IO data cursor. A degenerate *View* consisting of a single *Aggregate* typically suffices for SEQ syntaxes, and also occurs in many RA syntax elements. Constructing a SEQ import or export binding to a *View* amounts to instantiating the member *Aggregate*s with the IO data handle. Constructing a RA binding to a *View*, in particular, one containing a dynamic (i.e., variable length) *Aggregate*, is more complex, and is described next.

A *View*'s attributes (in particular, its size) are derived from its first *Aggregate*.[6] For a *View* consisting of only fixed–size *Element*s, the size is simply the statically computed sum of the first *Aggregate*'s *Element* sizes. RA imposes the additional constraint on the syntax specification that the size of an imported dynamic *View* must be specified by a *Semantic Variable* or *Field* labeled "size" contained within the fixed prefix of the first *Aggregate*—the presence of encoded size information is ubiquitous in existing syntaxes intended for RA. To utilize this field, Xync generates an auxiliary member class and static–member *Size* function. The auxiliary class duplicates the fixed prefix up to and including the size field. On import, the Size function instantiates the fixed size auxiliary class (its buffer requirements are known since it is fixed) and reads its size field. Along with the alignment values from the enclosing *Xynion* or *SubXynion*, the size value is passed to the underlying IO data buffer to ensure its contents can be bound by the *View*'s static and dynamic *Element* accessors. The size attribute also facilitates scanning a syntax instantiation containing variable length syntax elements without constructing their dynamic suffixes.

Since all *Element*s of the *View*'s alternative *Aggregate*s occupy the lexical scope introduced by the *Xynion* or *SubXynion* containing the *View*, the *Element*s must be uniquely labeled. In an RA syntax, an *Element* in one alternative can reference value–type *Element*s or *Element* attributes in the same or other alternatives, with the restriction that during initialization, only constructs specified lexically previous to the *Element* being instantiated may be dereferenced for a value.

Xync generates a *View* by emitting each alternative *Aggregate* in the order listed in the Xyn specification; the running offset computation is reset to the *View*'s base offset at the start of each alternative's generation. For an RA *View*, the attribute computations are emitted next. All alternative *Aggregate*s are initialized in their order of appearance in the Xyn specification, when the enclosing *Xynion* or *SubXynion* is initialized.

[6]RA constraints dictate that the size of a *Xynion* or *SubXynion* that is to be imported must be available as an attribute of the class defined by the Xyn construct: either as a compile time value for a fixed–size construct or as a runtime initialization value available through a static–member (i.e., class) function, i.e., a function that can be invoked without instantiating the class.

4.3.4 Choice

The Xyn *Choice* construct introduces branching into the syntax, and is one possible body of the *Xynion* and *SubXynion* constructs (the other is a *View*). The *Choice*, with its predicated branches, approximates a discriminated union. The *Choice* construct is semantically distinct from the *View*, since exactly one *Choice* branch is valid per syntax instantiation. Furthermore, a *Choice* is considered to be a dynamic construct by default, since its branches are assumed to have different sizes. Xyn restricts the *Choice* branch type to be a single, labeled *Element* (excluding *Semantic Variable*, but including Blex *Field*, *Xynion* reference, *SubXynion* and *Sequence* types), mainly as an artifact of using the label as the basis for branch selection and instantiation mechanisms. An example of the internal use of labels is Xync generating an enumerated *discriminator key* type for a *Choice* construct using the branch labels for its members.

Each *Choice* branch must be prefaced with a syntactic or semantic predicate, except the last branch, which becomes the default branch without a predicate and makes the entire *Choice* construct optional syntax *Element* when left empty. Xyn restricts a *Choice* to employing semantic or syntactic predicates exclusively, not a mix. Hence, Xync combines the predicates into a *syntactic* or *semantic* query that discovers the valid branch on import; analogous to lookahead disambiguation in a conventional programming language parser. Xyn exploits the nature of IO data transfer syntaxes by constraining a syntactic query to a fixed set of literal fields in the *Choice* prefix. To improve runtime efficiency, adjacent literal fields of a predicate are concatenated into a single literal. A syntactic query is implemented as combination of a Blex field–binding to retrieve the field's value and a C++ *switch* statement to validate the value and translate it to the appropriate branch key. A semantic predicate does not depend on the IO data prefix, so the semantic query is implemented as a sequence of conditional statements, each corresponding to a branch predicate; the associated branch key is returned for the first condition evaluating to true.

Choice initialization differs for import and export operations. On import, the syntactic or semantic query is invoked as part of the *Choice* object initialization, and the query result is stored in the *Choice*'s key field. Then the branch accessor corresponding to the query result is initialized (i.e., bound to the IO data) in its type specific manner and stored in an embedded union–like structure shared by all branches (since only one can be valid per *Choice* instantiation). A user can retrieve a branch accessor reference from an initialized *Choice* by first requesting the key value, then invoking the *Choice*'s corresponding per branch–type conversion function. On export, the *Choice* is initialized to a special "empty" state that mainly serves to record the underlying IO data buffer cursor. When a branch is selected for the *Choice* (using the same branch–specific conversion member function as for import), the *Choice* is automatically re–initialized to the requested type. Once the *Choice*'s discrimiator key is set to a value other than "empty," invoking a branch accessor request not matching the key is an error.

A *Choice* construct requires various size attributes. The class attribute *static_size* is that required for determining the *Choice* branch: the size of the query for import (i.e., zero for a semantic query and the field size of the syntactic query) and zero for export. As is the case with the *View* construct, RA import requires the (syntax) size of a *Choice* be available prior to the *Choice* initialization to ensure the underlying IO data buffer meets the binding requirements. The *Choice*'s static–member *Size* function invokes the *Choice*'s query to discover the branch type, then invokes that branch's static–member *Size* function to discover the actual size. Once initialized for import or export, the *Choice* adopts the size of its instantiated branch, which is still zero for the "empty" export state.

Xync generates a *Choice* construct by first generating each branch type not defined elsewhere (e.g., a *SubXynion* branch is specified inline with the *Choice* and defined as a member class in the implementation). Xync then generates the enumeration type defining the branch key, the semantic or syntactic query, and a type conversion operator for each branch. The per–branch type conversion operator returns a reference to the internally instantiated branch accessor (when valid). All member branches share the lexical scope introduced by the enclosing *Xynion* or *SubXynion*, including type parameters such as alignment.

4.3.5 Syntaxes, Xynions, and SubXynions

A Xyn *Xynion* or *SubXynion* implements the instantiation context for a syntax message or message element. Each construct introduces a new lexical scope — by introducing a new

C++ class definition — and provides a consistent interface to type and runtime instantiation parameters used by its constituent *Choice* or *View* body. Factoring the instantiation context from the member accessors forms a key nexus in the Xyn and Xync architectures, and significantly simplifies the implementation, since it allows the *Choice* and *View* constructs to be used independent of their location in a Xyn specification hierarchy.

A *Xynion* is a Xync generated C++ class defined at *Syntax* level — the implementation's enclosing C++ *namespace*, named by the Xyn specification's *Syntax* label, such as *IPv4Net*. A *public Xynion* corresponds to a transfer syntax message, while a *protected Xynion* is a message component, defined separately either for reuse or specification clarity. A key feature of both *public* and *private Xynion*s is support for user–defined parameters, allowing references to the *Xynion*'s instantiation context by elements defined within the *Xynion*'s context. Xync creates a member variable for each parameter, and adds a parameter to the *Xynion* initialization function. Other elements defined within the *Xynion*'s lexical context can reference the parameter's run–time value.

The IO data processing layer instantiates a *public Xynion* using the *Xynion*'s static–member *MakeAccessor* function, which takes a handle to the IO data object and returns an initialized *Xynion* accessor bound to the transfer syntax instance. Examples of *public Xynion*s include an IP *datagram*, and an MPEG *Sequence header* or *Slice*.

A *protected Xynion* type is defined similarly to its public counterpart; however, it is instantiated by a reference to its type from within another Xyn construct. The *Xynion* is parameterized at its point of instantiation (i.e., where referenced in the Xyn specification). Static context is passed through C++ type (i.e., template) parameters, while dynamic context is passed both through parameters defined in the *Xynion*'s specification and those declared in the Xyn specification, and realized as initializer arguments. Examples include the IPv4 syntax *Address* and *Option* types.

A *SubXynion* is a Xync synthesized C++ member class defined and instantiated "inline" (i.e., in series) with its adjacently declared *Aggregate Element*s. A *SubXynion*'s member name is its specification label, and its type name is derived by appending a *_t* to the label. Its qualified type name consists of that derivation prefixed by the lexical path to its definition, which consists of a top–level *Xynion* and possibly additional intervening

*SubXynion*s. The instantiated object binds to the underlying IO data buffer at the *Element* offset provided by its containing *Aggregate*, and provides its size information to the *Aggregate*, like other Xyn *Element*s such as a *Field*. The lexical context of the *SubXynion* definition is passed to the *SubXynion* instantiation as a mandatory initialization argument and is available to the *SubXynion*'s constituent *Element*s for their initializations. While nested lexical scopes are not natively supported by the C++ implementation language — a function of a member class has no inherent access to the enclosing class's members, i.e., the nesting is at the class domain, not the object domain — it is natural in Xyn to specify things like the termination conditions of a *Sequence* in terms of the *Sequence*'s enclosing scope.

4.3.6 Sequences

A Xyn *Sequence* is an instantiated wrapper class syntactically related to the *SubXynion* through the use of parenthesis to introduce a new lexical scope to its constituent elements. The *Sequence* declaration is augmented with suffixed attributes denoting a particular style of sequence; styles include *closure*, *fixed*, and *termination token*.

A Xyn *closure* is specified with the syntax *label:(element_type)**. A *Sequence*'s element type is a *Choice* construct, which must be optional (i.e., have an empty branch). For specification readability, a non–optional *Choice* is allowed in the syntax specification, but is automatically converted to optional by Xync.

On import, the *Sequence* is lazily initialized by repeatedly instantiating its *Choice* element as its elements are accessed, and advancing the IO data cursor the size of the branch recognized by the instantiation. This assumes elements are accessed in order of appearance in the IO data, which is a reasonable constraint even for RA (note that the initialized prefix of a *Sequence* can still be access randomly). An alternative implementation is available that aggressively parses the *Sequence* during initialization, allowing transparent RA use; however, in the current implementation, there is a performance benefit to initializing the element close to its point of use. The *Sequence* terminates when the reported element size is zero (i.e., the selected *Choice* branch is the empty "skip" branch). Initialized element accessors are available through the *Sequence*'s index operator (C++ `operator[]`). On export, the *Sequence* is initialized as an empty Xyn *Aggregate Element* (0–elements, 0–size). An element is appended by invoking the *Sequence*'s *NewElement()* function to return a reference to a "cleared" *Choice* element at the current *Sequence* index, instantiating the desired *Choice* branch type, and then explicitly calling the *Sequence*'s *Advance()* function. The explicit call to *Advance()* is necessary since the size of the appended element is not known until instantiated by the *Sequence* user. The requested *Choice* branch accessor is constructed in–place in the *Sequence*'s internal array, then the Advance function queries the initialized element for its size and advances the IO data handle that amount. SEQ uses a similar *Sequence* implementation, but provides a C++ *iterator* style interface — input iterator for import and output iterator for export — and hence does not allow access to any but the current *Sequence* element.

A Xyn *Fixed_Sequence* is specified using the syntax *label:(a)[n]*, where the element *a* is a fixed (at compile time) sized element and *n* is a fixed (at sequence instantiation) element count. Furthermore, the element size must be a multiple of its alignment requirement, ensuring alignment for all elements. Since the termination condition does not depend on the element, the sequence, including its element accessors, can be initialized for both import and export without examining the IO data. Initialization performance is enhanced since a single call to the underlying IO data buffering mechanism is required to bind the entire *Sequence*, which is then available for random access to its elements. This structure approximates a dynamically allocated C++ array, providing access via the *Sequence* index operator.

A Xyn *termination token Sequence* is specified using the syntax *(b)['EOS_literal']*, where the element *b* is of arbitrary type, and, while the terminating literal need not be a member of the element set, no member may have the literal for a prefix. This style of *Sequence* is implemented similarly to the closure style, but with a specialized query that tests for the *absence* of the EOS token. Once encountered, the token is consumed by default.

Xyn *Sequence*s are instantiated types — i.e., each style is implemented as a C++ template, and Xync instantiates the appropriate style's template with the sequence element type at the point of the *Sequence*'s specification. The parameters appropriate for the specialized *Sequence* type, i.e., element count or terminating literal, are passed to the *Sequence* initialization.

4.4 Summary

This chapter described the C++ implementation of the Xyn architecture and language described in Chapters 2 and 3. The implementation provides transparent access to the abstract syntax elements embedded in IO data to a layer's semantic operations through conventional C++ types and access idioms. Section 4.1 described the implementation architecture in terms of the *generation*, *composition*, *instantiation*, and *access* interfaces, shared among the implementation's elements. Sections 4.2 and 4.3 present the Blex and Xyn C++ implementations in a bottom–up fashion, describing each layer in terms of the above set of interfaces and how each layer builds on the facility provided by lower layers. The key contribution of the implementation lies in its demonstration that the flexible, layered IO data syntax specification language and binding architecture of Chapters 2 and 3 can be realized in a similarly layered, yet efficient implementation.

Chapter 5

Experience

The previous chapter described the general implementation features of each Xyn element, and their integration. This chapter describes an application of Xyn, namely the reimplementation of IO data access in the Click Modular Router Toolkit [Kohler et al., 2000] to use Xyn's generated constructs. The investigation examines both the esthetics and performance of Xyn's IO data accessors deployed in the Click environment. Section 5.1 introduces Click and provides the rational for using Click to evaluate Xyn. Section 5.2 describes how Xyn accessors were integrated into several Click elements, comparing the relevant source code. Section 5.3 benchmarks the Click modifications and then provides a collection of micro-benchmarks to explain the overhead of each Xyn feature added to Click. Section 5.4 concludes the chapter with discussion of the tradeoffs in using Xyn compared to conventional IO data binding techniques.

5.1 Click

Click is a software implementation of the Internet Protocol router specification written in C++. Click implements the IP layer and related protocols such as the Address Resolution Protocol (ARP) [Plummer, 1982] through a collection of simple–purpose elements that are combined into an IP router per a description (graph) written in the Click configuration language. Click's research contribution is in demonstrating the viability of decomposing router function into functionally orthogonal elements, while maintaining a high degree of efficiency expected of any IP implementation, but especially a network router. Although likely not the motivation for Click's modularity, an extreme example of the decomposition is the *DecIPTTL* element, which decrements the IP *time–to–live* field and recomputes the IP header checksum; a mandatory operation applied to every packet passing through the router.

Click's implementation utilizes C++features intended to facilitate modularity. Each router element is encapsulated in a class that inherits from a common *Element* base class. The base class provides a collection of virtual function interfaces that each Click element selectively overrides to implement its specialized function. While Click's modularity makes use of C++'s object orientation, the actual IO data access and processing code is essentially C – similar to that found in the canonical BSD TCP/IP implementation, which was designed with C as the target implementation language. The IP implementation's IO data access mechanisms exploit C's inherent coupling to the hardware architecture, including pointer casts and pointer arithmetic, and bit-field access to sub–integral sized IO data elements, that are succinctly expressed in compiler generated machine code.

5.1.1 Why Click?

Click provides challenges along several dimensions in which to evaluate Xyn: Since efficiency is a key metric for an IP router, Xyn's performance impact can be reliably measured (i.e., overhead can not be hidden by replacing fluff with fluff), and compared to known efficient code. Furthermore, since C is the prevalent IO data access implementation language — both for hand–coded implementation and generated mechanisms such as RPC IDL instantiations — Click provides good example of integrating Xyn into a typical IO data access environment. Additionally, Click's modularity requires the implementation of the Xyn language IP syntax specification be usable across a collection of Click elements, each of which only interacts with a portion of the syntax. Finally, IP requires RA processing, which introduces challenges over SEQ syntax processing. In particular, since Xyn was initially designed and prototyped for a SEQ syntax (an MPEG parser), the IP investigation has enhanced Xyn's robustness. In summary, Click provides a good test to evaluate Xyn's impact on code clarity and utility, and its ability to be integrated with a sophisticated framework.[1]

5.1.2 Click Configuration

The Click configuration used in the experimental evaluation is shown in Figure 5.1. This configuration is similar to that described in [Kohler et al., 2000], but replaces network interface elements with elements that simulate the network hardware. In particular, InfiniteSource generates a stream of Ethernet encapsulated packets.

5.2 Deploying Xyn in a Click Router

Since Click reads its configuration from a graph at startup, evaluating Xyn's performance in this context required renaming and modifying the Click modules of interest to use Xyn accessors, and then specifying the new module names in place of the original Click modules in the configuration graph. Three functionally distinct modules were used for this evaluation: *CheckIPHeader*, *IPGWOptions*, and *DecIPTTL*. *CheckIPHeader* is the initial element encountered on the IP layer processing path. This module parses the fixed (20 octet) IP header, validates various parameters, and checksums the IP header to test for corruption.

IPGWOptions contains by far the most complicated interaction with the IO data along the IP header processing path. The IO data cursor traverses the list of options, and for each option encountered, control is dispatched to the processing code for that option type. The options processing terminates when either the IO data cursor reaches the end of the header, or encounters an end–of–sequence option. Some of the options consist of a simple fixed–size aggregate of fields, while others are significantly more complex. An example of the latter is the timestamp option, which consists of a fixed header followed by a sequence of timestamps. Timestamp sub-options include recording the node address along with the timestamp, and "prespecifying" the list of addresses to indicate which nodes are to record timestamps.

In contrast to *IPGWOptions*, the *DecIPTTL* element provides an example of a fine–grained module; it decrements the IP header time–to–live field and adjusts the checksum

[1]An important aside: Click provides a full *user–level* implementation of the IP layer. Since Click's forwarding path and Xyn's IO data access mechanisms are independent of execution privilege, eliminating the complexities of in–kernel development greatly facilitated Xyn validation and benchmarking.

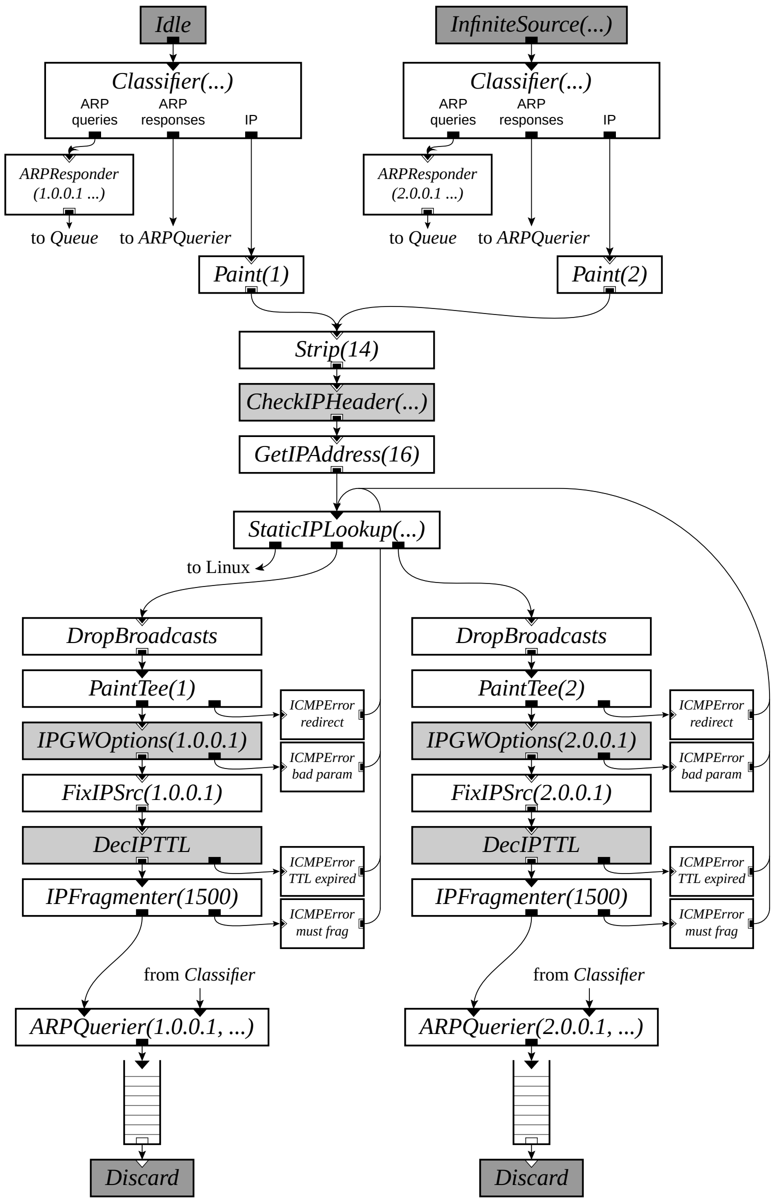

Figure 5.1: Click configuration used in Xyn evaluation. This configuration (and diagram) are derived from the configuration of [Kohler et al., 2000]. The light gray elements are the ones modified for the Xyn deployment, while the dark gray elements are replacements for the network hardware interfaces to allow Click to be evaluated out of the Linux kernel.

field to accommodate the new value. These three modules provide a representative sample of IO data access in Click; the details on the Xyn deployment are presented next.

5.2.1 Xyn Accessors in Click

The first required modification was to define and initialize the Xyn syntax level construct C++ *class IPv4Net::IP* instantiation, replacing Click's conventional pointer conversion from the *char*∗ raw data pointer to a *struct ip*∗:

```
click_ip *ip = reinterpret_cast<click_ip *>(p->data() + _offset);
```

becomes

```
XynBuf xynb(p->data()+_offset,(p->length()-_offset)*8);
IP ip0 IPv4Net::IP::MakeAccessor(xynb);
IP* const ip(&ip0);
```

Xyn expects a particular buffer interface, and since modifying Click throughout to use this interface was beyond the scope of the evaluation, each use of Xyn in a Click module requires an adapter from a raw storage pointer and length to the Xyn buffer interface; as shown in the first line. The Xyn accessor is constructed in the second line, and a pointer to the accessor is defined and initialized in the third line.

To a large extent, Xyn's IO data accessors were transparently deployed in the fixed header processing of *CheckIPHeader*, or the field update in *DecIPTTL*. The Blex element encapsulation facilitates using the accessor as its equivalent integral (host) type, so the notation

```
ip->ip_hl
```

is semantically equivalent in Xyn field access and conventional C struct member dereference usage.

Modifications were also required for any field declared in the Xyn specification to be a Blex *literal* or a *Semantic Variable*. In Xyn, literals are translated into a boolean — true when the field value matches the specification, otherwise false — so the test for equality in a conditional expression can be replaced with the boolean valued field label:

```
if(ip->ip_v != 4) ...
```

becomes

```
        if(!ip->ip_v) ...
```

Computation corresponding to the syntax is encapsulated in *Semantic Variable*s, so the computation specified in the IO data processing code (and associated variables) can be replaced by the appropriate *Semantic Variable*:

```
        unsigned hlen = ip->ip_hl << 2;
```

becomes

```
        unsigned hlen = ip->ip_hl_bytes;
```

The value of encapsulating this relatively simple computation in a *Semantic Variable* is twofold: first, the computation comes directly from the syntax specification, and second, the header length value is used in more than one place — in the Xyn reimplementation, the variable *hlen* is replaced by the *Semantic Variable* at each use.

The module most impacted by the Xyn deployment is *IPGWOptions*. The IP options syntax suffers from the desire to keep the IP header compact and compatible with hardware access requirements, while at the same time allowing flexibility in the range of option types and organization. Hence, the conventional option sequence parsing implementation utilizes obscure pointer manipulation to ensure efficiency while coping with complex syntax. In contrast, the Xyn IP specification encompasses the syntactic complexity, so the Xyn accessors present the options list and individual options as conventional host data structures, resulting in more readable processing code. Several examples are presented next. An example might consist of discontiguous lines of code; for example, to demonstrate both reading and writing a syntax element.

When evaluating the examples, it is important to keep two factors in mind. First, the Click authors likely focused little effort on the readability of the transformation code, since it is unrelated to demonstrating the viability of the modular router concept; and second, when compared to the fixed IP header processing, the options processing code readability is obscured by accommodating the possibility of a multi–byte option structure following a single byte option. This ordering results in the latter's fields falling on arbitrary host alignment boundaries, hence code must use techniques such as substituting the C `memcpy` function for any multi–byte syntax field read or write. With some effort, the clarity of the "conventional" implementation could be improved through the usual C macro and cast techniques. Thus the point of the examples is to demonstrate how Xyn's generated accessors automatically provide improved code clarity — both in simplified syntax field access and by allowing the processing code author to focus on the structure of the processing code.

The first example is the outer loop that iterates over the options list. The original Click code uses an integer index into the IP header data octet (byte) array (named by `oa`):

```
for (int oi = sizeof(click_ip); oi < hlen; ) {
  ...
  // otherwise, get option length
  int xlen = oa[oi + 1];
  ...
  oi += xlen;
}
```

which is advanced the length of the current option at the end of the options processing. The substitute Xyn implementation utilizes the built–in element count attribute of the *Sequence* type:

```
for( int oli = 0 ; oli < ip->ip_options.SizeElements() ; ++ oli ) {
 ...
}
```

Conventional access to the option is through an index into the byte array containing the IP header data. For example, the option type is accessed, and compared to the C preprocessor macro value corresponding to the timestamp option:

```
unsigned type = oa[oi];
...
} else if(type == IPOPT_TS){
```

Retrieving the Xyn *Sequence* element requires first initializing a C++ reference to the *Choice* construct embedded in the generated IP options syntax accessor, then querying the *Choice* for its type, and finally initializing a reference to the indicated type:

```
option_t & op( ip->ip_options[oli] ) ;
switch( op.WhichBranch() ) {
...
case option_t::TKey::ts :
  {
    option_t::ts_t & ts( op ) ;
```

Processing the timestamp option requires accessing several syntax attributes along with recording the timestamp and, optionally, host address in the timestamp list. The complete code to update the timestamp list is somewhat convoluted, due to the packed encoding and various possible sub–options, so rather than listing it in its entirety here, some example expressions are presented instead. First, accessing and updating the timestamp list pointer, which points to the position the next timestamp is to be recorded and is encoded as a 1–based index into the option octet data (i.e., consider the entire option as an octet sequence). After recording the timestamp, the pointer is advanced to the next position. The amount advanced depends on whether or not the timestamp list element includes the host address. The original Click implementation reads the current value into a variable `p`, and updates the value like this (in this case the timestamp list element includes an IP address and hence is 8 bytes):

```
int p = oa[oi+2] - 1;
...
woa[oi+2] += 8;
```

The Xyn–based reimplementation can access the pointer value through its Xyn specification name, and the size of a list element through a *Semantic Variable*:

```
ts.pointer += ts.list_elementOctets ;
```

The timestamp option header also includes four flag bits, two of which are used. When the least–significant bit is set, each timestamp list element includes the recording nodes IP address. When the next bit is (also) set, the timestamp list contains a list of "prespecified" addresses, indicating which nodes are to record a timestamp. The flag field occupies the low–order bits of an octet; the overflow counter resides in the upper four bits. The original Click implementation accesses the flag bits like this:

```
int flg = oa[oi+3] & 0xf;
...
} else if(flg == 1){
...
} else if (flg == 3 && p + 8 <= xlen) {
```

The Xyn based reimplementation specifies each flag bit as a boolean type:

```
} else if( ts.flags_isAddr ){
...
} else if( ts.flags_isPrsp && ts.pointer + ts.list_elementOctets <= ts.length +
1 ){
```

When the timestamp list reaches capacity, any further nodes encountered increment the four–bit overflow count (if the overflow count reaches 15, the next node encountered generates an error reply to the originating node). The original Click implementation uses shifts to read the field value, and must combine it with the flag field to update it:

```
  int oflw = oa[oi+3] >> 4;
  ...
  if (oflw < 15)
    woa[oi+3] = ((oflw + 1) << 4) | flg;
```

The Xyn generated accessor manages these details, and can be used like a locally defined variable:

```
  if( ts.overflow < 15 ){
    ++ ts.overflow ;
```

Finally, accessing the timestamp list element; in this case it includes the IP address. Recall the original Click implementation must accommodate fields not falling on integral boundaries (`ms` is the timestamp value):

```
  memcpy(woa + oi + p, &_my_ip, 4);
  memcpy(woa + oi + p + 4, &ms, 4);
```

The Xyn reimplementation must acquire a reference to the list element accessor at the index computed by the *Semantic Variable* `list_elementIndex`:

```
  typedef option_t::ts_t::list_t::avec_t list_t ;
  list_t::element_type &e(((list_t &)ts.list)[ ts.list_elementIndex ]) ;
  ...
  e.node.s_addr = IPAddress( _my_ip ) ;
  e.isNS = false ;
  e.time = ms ;
```

As seen with the list type in this example, Xyn's layering leads to long type names; however, the names are derived directly from the labels in the IP syntax specification by appending an "_t" to the label, and interference with readability can be ameliorated through the use of `typedef` typename aliasing.

implementation	*elements*		*characters*	*lines*
	dynamic	*all*		
Click	382	391	2249	88
Xyn	272	335	2852	89

Table 5.1: The results of the Unix *wc* command on the original Click IPGWOptions module and the version modified to use Xyn bindings. The modules were reformatted so that identifiers and operators are counted as program elements, but puncutation is not. The *dynamic elements* column does not include the count of the static scope resolution operator "::" nor the member selection operator ".", since these are computed offline. The Xyn version uses these extensively due to the layered implementation of the Xync generated bindings. The insignificant difference in line counts is a coincedence, while the Xyn version's high character count is partially due to the layered binding types and partially due to more expressive variable names.

5.2.2 Code Metric

As a coarse metric of the succinctness of Xyn's IO data accessors compared to conventional C–style access, the Unix *wc* (word count) program was used to compare the Xyn and original implementations of Click's IPGWOptions element. Specifically, the method that tests for and then processes any per–datagram IP options was measured for each implementation. First, all comments and empty lines were removed. Second, the program text was put in a canonical form designed so that identifiers (types, variables, and functions) and operators (control flow, function application, array indexing, arithmetic, and logical) are counted, while punctuation (parenthesis, commas, semicolons, braces) is not. Table 5.1 shows how Xyn accessors provide a succinct means to access complex IO data structures using expressions derived directly from the syntax specification. The required low–level mechanisms are wrapped in the accessor implementation, relieving the layer processing author from having to deal with C–style pointer arithmetic and bit manipulation.

5.3 Performance Evaluation

This section describes two collections of benchmarks evaluating Xyn's generated access bindings. The first collection measures the performance impact of the modifications to the Click router described in the previous section. The second is a collection of micro–

benchmarks measuring the costs in cycles and instructions of each Xyn feature layer (i.e., abstraction) used in the Click deployment.

5.3.1 Evaluation Environment

The systems primarily used for the evaluation are two Sun dual–processor workstations running the Solaris operating system. Since the duration of some of the benchmarks is longer than a scheduler time quantum, a multiprocessor system simplifies running the benchmark, since can it bind to one processor and not be preempted. The machine used for the Click benchmarks is a Sun Blade 1000 with two UltraSparc–III CPUs running at 750MHz. For the characteristics of processing the Click forwarding path, this machine is comparable to the Pentium III system used in the original Click evaluation [Kohler et al., 2000]. In particular, both have a 4–way set–associative level 1 cache that holds the relatively small program working set, and the purely integer computation with extensive IO restricts most opportunities to exploit the super–scalar features of the UltraSparc processor. Since the baseline Click performance on the Sun machine matches that reported in [Kohler et al., 2000], the confidence of the validity of the evaluation reported here is high. The micro–benchmarks were run on a Sun Ultra–60 with two 360MHz Ultra II processors. The absence of non–IO related computation in the micro-benchmarks puts too much pressure on the store buffer of the higher frequency processor resulting in extensive processor stalls and artificially high cycle counts.

Xyn generated code is accepted by at least KAI C++ 4.0f (KCC), GCC 3.3 (g++), Sun C++ 5.5 (CC), and Metrowerks CW 8.3 (mwcc). Previous releases of Sun C++ and GCC were unable to parse some of the template constructs used by Xyn; in particular, the layers of templates and template techniques used to compute access parameters in the type system to ensure minimal runtime overhead. Of these compilers, KCC and g++ allow precise control over inlining, in particular aggressively inlining specifically those functions marked with the `inline` keyword. Compilers without this degree of control either do not inline the marked functions aggressively enough, or take interminably long to compile each module due to attempting to inline all routines. Finally, KCC generates an intermediate representation in C, and uses the host's C compiler to generate the machine code. For these

configuration	*fixed header*				*with timestamp option*			
	cycles		*instructions*		*cycles*		*instructions*	
Click	983		804		1335		1170	
CheckIPHeader	1037	(+54)	848	(+44)	1397	(+62)	1219	(+49)
DecIPTTL	1071	(+34)	873	(+25)	1433	(+36)	1241	(+22)
IPGWOptions	1147	(+76)	954	(+81)	1749	(+316)	1527	(+286)

Table 5.2: Measuring Click's processing a 20 octet (fixed) IP header and a 32 octet IP header that includes the timestamp option. The first row shows the cycle and instruction counts of Click using the stock router elements. The next three rows show the counts as elements CheckIPHeader, DecIPTTL, and IPGWOptions that have been modified to use Xyn's generated IO data accessors replace (cumulatively) their corresponding original Click elements. The numbers in parenthesis are the overheads from introducing the Xyn accessors into the particular element.

experiments, the Sun C compiler is the target back-end compiler, and it does a better job of instruction scheduling and data layout for the SPARC hardware than GCC does.[2]

In summary, Xyn challenges compilers: extensive use of templates (in generated code) requires standards conformant compiler, fine–grained layering demands aggressive inlining, and IO intensive demands instruction scheduling. KCC with the Sun CC back-end satisfies these criteria with a standards conformant C++ front-end, aggressive and controllable C++ optimizations, along with instruction scheduling of the optimized native C compiler. As an indication of the importance of the compiler, there are two orders of magnitude difference in the cycle count of the unoptimized and optimized builds of the micro–benchmarks.

5.3.2 Click Benchmark

Figure 5.1 shows the test configuration for this benchmark. The evaluation consisted of the stock Click router, and this router with the with the original *CheckIPHeader*, *DecIPTTL*, and *IPGWOptions* elements replaced by versions using Xyn generated accessors. The modules to be evaluated were chosen for their distinct features. *CheckIPHeader* parses the fixed

[2]As an aside, while Click's primary implementation target is Linux/x86 (using the g++ compiler), when I started developing Xyn there was no C++ compiler comparable to KCC available for the x86 platform. Eddie Kohler, Click's primary architect, has been very accommodating in helping me port Click to the Solaris–KCC combination. In particular, modifying Click source accepted by g++ to be accepted by both KCC and g++.

IP header, reading many fields and has the highest per–element cost on the forwarding path for the original Click implementation. In contrast, the relatively simple *DecIPTTL* ensures time–to–live field is at least two, decrements it, then reads and computes a new checksum using a specialized algorithm consisting of six arithmetic operations, and records the new checksum in the IP header. Finally, *IPGWOptions* introduces a branching access structure in order to parse and update the complex IP options syntax.

One additional module exchange was introduced for each run, so that both the overhead contributed by each module and the total overhead could be measured. For each configuration, two runs were evaluated. The first with the basic 20 octet IP header, and the second with an additional 12 octet timestamp option, that includes the fixed timestamp option header, a recorded timestamp, and space for *IPGWOptions* to append its timestamp. For each run, 100,000 packets are emitted by *InfiniteSource*, and a counter element records CPU performance counter data in the packet annotation fields before passing the packet to the Click Classifier element. An accumulator element located after IPFragmenter increments a counter for each arriving packet and accumulates the difference between the recorded performance counter values from the current counter values. All packets for a run are identical, ensuring minimal variations from operations such as route lookup — minimizing Click overhead in this case since the route lookup operation caches routes for recently seen destination addresses.

The results are shown in Table 5.2, with each row labeled with the additional Click element modified for that test. For reference purposes, [Kohler et al., 2000] reports the cost of processing the fixed IP header for *CheckIPHeader* as 320 cycles, *DecIPTTL* as 84 cycles, and *IPGWOptions* as 45 cycles. Given the close correspondence of the overall performance of the implementation in the reference and the stock implementation evaluated here, these figures are applicable to the current evaluation. Additionally, [Kohler et al., 2000] reports on techniques they developed to substantially reduce per–element overhead, such as a tool to automatically eliminate the virtual function calls occurring at each element packet ingress and egress from a Click configuration. Hence, any overhead introduced by Xyn accessors would have a greater impact on performance in a Click router enhanced using these tools.

The intention in deploying Xyn accessors in Click was twofold: first to validate their

feature	*construct*	*data*	*cycles*	*instructions*
External Buffer	static	IP	30	51
Fixed Header	static	IP	68	73
Semantic Variables (Init)	Sem. Var.	IP	69	77
TS Elements	static	TS	152	154
TS Time–only List	Fixed Seq.	TS	182	217
TS Semantic Variables	Sem. Var.	TS	229	229
TS List Type Selector	Choice	TS	328	305
IP Option Selector	Choice	TS	349	326
IP Options List Wrapper	Choice	TS	374	354
IP Options List	Sequence	TS	449	441
IP Options, Empty List	Sequence	IP	124	111

Table 5.3: Micro–benchmark measuring field access costs per Xyn feature by converting between syntax and host types for each of the IP header fields. Initialization and accessing a *dynamic* construct such as a *Choice* or *Sequence* requires runtime context. An IP in the data column indicates the 20 octet fixed IP header was accessed, while TS indicates a 32 octet header consisting of the fixed header suffixed with a timestamp option containing two timestamp fields. The counts include initializing the Xyn accessor according to the feature, which typically dominates the cost of accessing any fields in the current implementation.

usability in an existing, realistic context, and, second, to validate the performance results of the collection of micro–benchmarks created during the development of Xyn. Since the Xyn–accessor induced overheads incurred in the Click evaluation correspond to those of the micro–benchmarks, the discussion of the overheads is deferred to the detailed evaluation of the next section.

5.3.3 Xyn Feature Micro–benchmarks

The Xyn micro–benchmark presented here is a field–by–field copy of an IP header, using generated import and export accessors including *Sequence* and *Choice* constructs for copying IP options fields, when present (see Appendix A for the syntax of an IP header). The source and target IP access objects are automatic variables defined and initialized inside the benchmark routine. The number of syntax fields to copy is 14 in the case of the static IP header and 28 in the case of the timestamp option. The number of Xyn accessor elements to initialize ranges from 22 in the case of the static IP header to 52 in the case of the IP option list accessor implemented as a Sequence (as was used in the Click benchmarks, above). The

Xyn field (Blex) accessors and *Semantic Variable*s require one or two object members to be initialized, while the *Xynion*, *Choice*, and *Sequence* navigation constructs require several object members be initialized.

Along with the initialization and copy costs, each benchmark result includes the overhead of accessing the performance counters, a function call to the copying routine, and construction of the Xyn import and export buffer objects from the passed in pointers to the IO data arrays. Each result presented in Table 5.3 is the raw result minus the counter overhead, divided by 2 (i.e., the average of import and export accesses). The function call overhead and buffer construction are assumed to be negligible, since the call arguments are passed in registers and the buffer initialization is performed inline (i.e., without a function call).

The per–feature benchmark tests mainly vary in the portion of the Xyn specification of the IP syntax required to evaluate a feature. Due to Xyn's layered nature and terse syntax, this was accomplished by taking the full IPv4 specification, then peeling off layers of the IP Options specification and renaming the result until only the fixed IP header remained (in the order opposite of that used to present the results in Table 5.3). The entire collection of specifications reside in a single file and were processed by Xync into the corresponding collection of C++ accessors. The IP syntax instantiation used for the first set of benchmarks is the plain IP header, the second set is passed the header appended with the timestamp option, while the final evaluation of an empty options list is again passed the plain header.[3]

Static Constructs

This section examines Xyn accessors operating on the fields of the fixed IP header. The Xyn constructs involved are a public *Xynion* corresponding to the IP header and implemented as a C++ class container for the per–field accessors, and the collection of per–field Blex access objects aggregated into a Xyn *View*. As described in Chapter 4, the *Xynion* buffer field is initialized with a reference to the IO data and various other attributes; in turn, each of the Blex field access objects is initialized with a reference to the Xynion's buffer, and, in the

[3]In fact, all except the last test could have been passed the timestamp options instantiation, since the first set of tests would simply ignore the fields beyond the fixed header.

case of literal fields (i.e., those defined in the syntax with literal values), the literal value. Keep in mind, that in order to present each syntax field with the same access interface (e.g., assignment to or from its name) as its corresponding host programming language primitive type, the Blex field must be implemented as an *object* having a type conversion operator for use as an *rvalue* and an assignment operator for use as an *lvalue.* Being a distinct object implies the accessor must embed a reference to its context (e.g., enclosing *SubXynion* or *Xynion*), in order to gain access to the IO data buffer to which it implements a binding. This per–field reference turns out to be the key factor impacting Xyn's current performance.

The first test *External Buffer* is actually an adaptation of the generated accessor for the fixed IP header. The *Xynion*'s embedded buffer field is replaced with a bare pointer, initialized to the address of the IP IO data. In this case, the C++ compiler (KCC) is able work through the Xyn and Blex layers to produce machine code to access a field that is essentially identical to that produced for dereferencing a pointer into a C struct. This amounts to loads and stores at a statically computed offset from the IO data base address, and shifts and masks to extract any non–integrally sized elements. Hence, this result is comparable to the conventional C implementation of IP header processing, and is baseline for the remainder of the performance measurements.

Once the *Xynion*'s bare pointer is replaced by an embedded buffer structure — even one as simple as an object holding a pointer to the actual IO data as its only member — performance anomalies begin to creep in. In the *Fixed Header* test, the instruction count is elevated by a store of each Xynion member state, in spite of 1) the saved state consisting of the identical pointer value along with several literal values, 2) accessors being defined on the stack, and 3) the compiler produced code never referencing the saved state. Aside from the sequence of per–field store operations, the computation is identical to the previous test, with IO data load and store targets occurring at statically computed offsets from the IO data base address. Since there is only one slot for a load or store operation per instruction group (taking one of the two integer instruction slots), the cycle count increases more than the instruction count when the superfluous store operations outnumber the integer instructions the scheduler can pair with the stores. Ultimately, the pipeline scheduler stalls when the store instructions fill the processor's store buffer.

As to why the compiler generates these stores: In the *External Buffer* test, the *Field* accessor objects hold a reference (implemented as a const pointer) to the *Xynion*'s bare pointer to the IO data, while in the *Fixed Header* test the *Field* accessors instead hold a reference to a *Xynion* structure member, which, in turn, holds the pointer to the IO data. Compiler optimizers track pointer chains, but perhaps the intervening structure disables this heuristic. Interestingly, this behavior is consistent across all the compilers discussed above. The same superfluous store anomaly also appears when a *Semantic Variable* is added to the Xyn specification used for *External Buffer*, but only if it is used in the IO data processing. A *Semantic Variable* state consists of a pointer to the enclosing *Xynion* or *SubXynion* lexical context; perhaps disturbing the optimizer's pointer analysis heuristic.

The next test, *Semantic Variables*, is the addition of two unused *Semantic Variable*s to *Fixed Header*. The slight additional overhead is from storing the initialized member state to the stack (again, superfluous). *TS Elements* directly appends the IP timestamp option's fixed header syntax and two timestamp fields to the Xyn specification of the IP fixed header. The superfluous store anomaly persists with the additional fields. In this case, the costs are exacerbated by the compiler generating single–word stores for each state element — but only for the export argument — rather than composing adjacent fields into registers for double–word stores (exploiting the UltraSparc 64–bit architecture). Oddly, previous and subsequent tests did not suffer from this anomaly.

Dynamic Constructs and Layering

The second collection of micro–benchmarks evaluates the performance of Xyn's dynamic constructs, *Choice* and *Sequence*, using sample data consisting of the fixed IP header and a timestamp option containing a timestamp list of two timestamp–only fields (i.e., the complete timestamp option syntax includes a *Choice* of either a list of timestamp+address elements or a list of timestamp–only elements).

The first test of this set, *TS Time–only List*, evaluates a Xyn specification consisting of the fixed IP header and a modified timestamp option that contains the option header and a list (Xyn *Fixed Sequence*) of timestamp–only fields — the *Choice* of list types has been elided. The processing of the static IP header and timestamp option header is identical to that of the previous test, where IO data fields are accessed at a static offset from the base address. Accessing the dynamic *Fixed Sequence* introduces the next major source of overhead. As described in Section 4.3, Xyn's *Sequence* and *Fixed Sequence* are created lazily by default, with the current index's element accessor constructed in a cache; the assumption being that the sequence will be traversed once. In theory, this design should give the optimizer the opportunity to iterate across the timestamp list, while keeping the sequence and element state in registers and recomputing the (relatively few) index–sensitive values with each iteration. In practice, the sequence and (currently indexed) element state are stored to, and subsequently loaded from, stack addresses. The addresses of the imported and exported IO data are then retrieved from the corresponding element state fields, rather than as offsets from the base IO data pointers. While the stack addresses are at static offsets from the frame pointer, the loads at the beginning of each instruction block and stores at its end cause significant overhead in the timestamp list processing. Two possible causes for not achieving the theoretical optimization are: 1) the combined state of the two sequences and the user loop context simply overflows the capacity of a single UltraSparc register window, which is slightly less than 32 integer registers;[4] and 2) the optimizer does not unroll the loop and speculatively pursue loop interior conditional branches to an extent necessary to expose the potential for subsequently collapsing each branch through redundant code elimination. Section 5.3.4 includes suggestions to reduce the amount of Xyn construct state and branching.

TS Time–only List computes the list length using a simple calculation consisting of subtracting the option fixed header length from the option length field value and dividing this difference by four (i.e., the length of the timestamp list element). The next micro–benchmark, *TS Semantic Variables*, replaces this computation with one composed from several *Semantic Variable*s operating on the timestamp fixed header fields. In contrast to the previous micro–benchmark, this computation includes a denominator that depends on the timestamp list element type, which is not known until the header field accessors are initialized with the IO data, and must be used for the complete timestamp option. Thus the compiler produces a much more complex computation, including accesses to the header

[4]An option for the back-end C compiler instructs that floating point registers be used in case of integer register overload, but this never occurred in these benchmarks.

fields and an integer division, and executes it once for each list (i.e., import and export). Additionally, each *Semantic Variable* adds one field to the gratuitous state stores (i.e., stored values never accessed).

The next three micro–benchmarks, *TS List Type Selector*, *TS Option Selector* and *IP Options List Wrapper*, each introduce a *Choice* construct. As discussed in Section 4.3, a *Choice* contains storage initialized to whichever branch is selected at runtime. While the *Choice* runtime type is automatically determined during initialization from type information in the IO data or program state on the import and update paths, the export path is complicated by deferring this specialization to when the IO processing accesses one of the *Choice* branch types. At this point, the *Choice* automatically "reinitializes" itself to the appropriate type. The current implementation utilizes the C++ *inplace new* operator, which constructs the requested object type at a memory location specified by a *pointer–to–void* (i.e., untyped pointer), supplied by the *Choice*. Since type information is lost with the initial conversion to pointer–to–void, the compiler optimizer must conservatively evaluate pointers and aliases, and can no longer optimize based on distinct types representing distinct objects. Thus, while the overhead of the actual *Choice* query and (re)initialization is on the order of a few instructions, the downstream cost is significantly higher.

The micro–benchmark *TS List Type Selector* introduces a *Choice* construct to select between variants of the timestamp option list element. One type of list consists of elements having a *time* field, while the other type's elements also include an *IP address* field for the timestamp recording host's address. The effect of employing the list type selector is that, rather than accessing the fixed–part IO data fields to be copied using the base address offset by a static value (composed in single load or store instruction), the IO data pointer buffer address is loaded from each field state, indexed into to get the IO data address, then the load or store is issued using this address. The fact that this address is identical for all import or export fields is now lost on the optimizer.

The remaining micro–benchmarks (two additional *Choice* constructs, and the complete IP accessor with a timestamp option *Sequence* and an empty *Sequence*) complete the analysis of the costs of layered abstractions in the current implementation. The causes of the overheads introduced are similar to those described already. The next section discusses the tradeoffs arising from the implementation and some candidate techniques to reduce the overhead.

5.3.4 Insights

When the IO data syntax consists of a simple aggregate, such as the fixed portion of the IP header, Xyn accessors can replicate the performance of conventional access techniques implemented in C. In this case, Xyn's functionality and performance approximate that of USC [O'Malley et al., 1994]. As navigation constructs are layered to implement the full IP header specification, overheads increase to the point that IO data access costs become a significant contributor to the total cost of processing a packet in the Click router benchmark.

The collection of micro–benchmarks discussed in the previous section illustrates the impact of each layered feature on the optimization achievable by a competent compiler tool-chain,[5] and leads to the ultimate question of if there is an inherent constraint in the approach that precludes maintaining the high degree of abstraction while achieving the goal of performance equivalent to hand–coded C accessors. Specifically, the question is if the extent of branching and pointer aliasing introduced with the complex syntax navigation constructs precludes the optimizer eliding the abstraction–enabling per–field access object state like it did in the simpler static micro–benchmarks. While this question remains to be resolved, there are a number of straightforward enhancements that can be made to Xyn's code generator to reduce overheads and simplify further analysis.

In general, these techniques can be describe as one of: 1) Moving state into the type system (e.g., by substituting a C++ template parameter), thus eliminating it as a point of variation in the runtime code generation; 2) Eliminating superfluous (unused or duplicated) state by further specializing Xync's code generation to elide state variables that are not used; and 3) Eliminating superfluous computation, also by specializing Xync.

The problems addressed by category 1 techniques largely arose from concern about code–bloat resulting from instantiation of many types with only slight variations. In this case, the concern was misguided: since only by eliminating the variations from the runtime

[5]While the intermediate representation and back-end optimizations (e.g., instruction scheduling) shared with the C compiler tool-chain are mature, it would be a stretch to claim maturity for any C++ compiler, since only in the past few months have mainstream C++ compilers begun to offer even nearly full support of standard C++ *syntax*. Heuristics to detect and optimize common abstractions have yet to appear.

code can the compiler fully optimize the remaining code (e.g., by eliminating state or operations that then become, or are exposed as, superfluous). Since the optimizer is inhibited by navigation runtime code complexity, a general technique is to move constant values examined by runtime conditionals into the type system. In particular, the access mode field present in every Xyn navigation construct is actually only referenced in the *Choice* construct to distinguish between update and export initialization.[6] Since a Xyn accessor is declared and defined at its point of use, moving this attribute into the type system (as a parameter to each Xyn navigation construct template) eliminates a runtime branch occurring in the midst of initializing a *Choice* object and its nested branch accessor. For another example, Blex literals (i.e., syntax fields with fixed values) currently store the literal value as an object member. Since the literal value is expressed in the host–domain and is typically an integral type, it can become a Blex template parameter. This eliminates half of a literal's per–element state and associated access costs, leaving only the IO data reference, common to all Blex types.

Xyn navigation constructs have several internal fields including references to both lexical context (i.e., the lexically enclosing Xyn construct), and the IO data buffer that is the target of the construct's IO data access bindings. *Semantic Variable*s and initialization arguments can make use of the lexical context to access the enclosing construct's state. If the nested construct uses neither of these features, then the lexical context handle is superfluous and can be deleted from the state. The Xyn architecture also supports reaching through a nesting level into the context of an outer enclosing construct, but only from another Xyn navigation construct. Both of these conditions can be detected by examining just the local context for *Semantic Variable* or Xyn constructs; when the construct at issue contains only Blex accessors, it always safe to remove the context handle. This optimization is primarily of interest for a *Sequence* element type, where the removal of a single field could provide a noticeable performance improvement.

The need for the final optimization category is exemplified by the Blex field writing mechanism. In general, in order to update a syntax field, the integral IO data buffer element(s) containing the syntax field must be read, the appropriate bits replaced by the

[6]Distinguishing between import (read–only), and the update and export modes is accomplished using the const attribute on object members.

new value, and the updated integral element(s) written back to the buffer. In many cases (e.g., almost all fields of the IP header), the syntax field corresponds to a host integral boundary, and hence only the store of the new value need take place. Eliding the read and merge eliminates about 80% of the write cost. Blex's bit access layer already implements a mechanism to determine the minimum width integral access for a syntax field (to minimize the overhead of endianness reversal); adapting this to detecting integral access writes in a type (i.e., static) computation is straightforward.

5.4 Summary

Clearly, there is a tradeoff in using the current implementation. On the one hand, the type–safe, abstract interface to IO data syntax significantly improves IO processing program esthetics, and by eliminating the tedious and error prone coding required to access intricate IO data syntax, increases the potential for robust implementations. On the other hand, the overhead introduced when exploiting the full generality of Xyn expressiveness constrains the applicability of the current implementation. An argument in defense of the approach taken with Xyn is that this is still an immature implementation. The analysis of the previous section suggests the elevated cycle counts from each feature layer added could be eliminated if the compiler optimizer could better evaluate the pointer manipulations that currently cause it to conservatively operate on each IO data element through the element's access object, rather than discarding this state that is never modified once initialized. Several straightforward changes to the Xyn code generator to address performance were suggested in the previous section.

Furthermore, even with its current overheads, Xyn is useful for applications less demanding than IP router environments. For example, client–node network stack, and user–level analysis or generation of IP data can benefit from Xyn's abstraction. Sequential access processing utilized for syntaxes such as MPEG requires less flexibility than the random access required by IP — the additional constraints imposed on the use of the accessors eliminates some of the generality restricting the optimizations the back-end compiler would perform. Finally, much like improvements to a general purpose language compiler can be exploited by recompiling the unmodified program source, performance improvements in the

Xyn code generator can be exploited by regenerating the access binding code from the Xyn specification, and recompiling the unmodified IO processing program source.

Chapter 6

MetaXyn

The previous three chapters described the IO data syntax specification language Xyn, the IO data access bindings generated by Xyn's compiler, Xync, and experience deploying the bindings in the Click router. These chapters focused on the *intra*–layer aspects of IO data binding; that is, binding a single layer's semantic operations to the syntactic elements embedded in an IO data message delivered to, or produced by, the layer. This chapter explores the *IO data access* related interactions resulting from the composition of layers into a subsystem providing a complete IO service for applications. In particular, the chapter addresses the open issues of Section 1.2 of the Introduction, which observed that while the hierarchically layered organization of the IO processing system is motivated by the software engineering principle of *separation of concerns*, in reality, the layers can not be entirely isolated. Instead, the IO processing layers are effectively bound through the IO data passing through the layer hierarchy, and hence interactions between layers are an unavoidable side effect of each layer imposing its local constraints on the shared IO data.

For example, [Clark and Tennenhouse, 1990] describes how isolation between layers leads to the network layer "packetizing" application data for transmission without regard for the application data boundaries. As a result, a network packet delivered to the receiving application endpoint(s) must be coalesced with other network packets (some of which might be delayed or lost) before any of the data can be provided to the application layer for consumption. Typically, the application is programmed to operate on a contiguous data element, which requires the data fragments to be copied into contiguous storage, adding uncessesary overhead. The solution proposed, *Application Level Framing* (ALF), requires the network layers to share information about the maximum message size with the application (actually, presentation) layer, so that each produced application data unit can be integrally packaged for transmission. The result of ALF is application data units delivered intact to the receiving application endpoints, where they can be passed to the application and processed immediately.

In general, when an inter–layer IO data access conflict arises as a result of a mismatch among intra–layer access constraints, the interfering layers can usually be isolated by copying the IO data from the upstream layer's buffer to a new buffer for the downstream layer; however, copying IO data leads to inefficient use of the memory hierarchy, potentially limiting the performance of IO intensive applications. Section 2.2 of the Architecture Chapter introduced a meta–layer protocol called *MetaXyn* to manage the *inter*–layer exchange of *intra*–layer attributes made available by Xyn's compiler, Xync, that can be utilized to avoid or at least minimize the copy overheads resulting from incomplete layer isolation.

In the case of the ALF example, Xync exposes the intra–layer *contiguity* attribute, which describes the integral message size range possible for the layer's users, i.e., that available after adjusting for framing, e.g., a header prefix. The ALF algorithm (implemented within the MetaXyn framework) utilizes MetaXyn to propagate the contiguity data through the layers, starting with the network device layer (the MetaXyn origin, in this case). At each layer, the algorithm computes the intersection of the contiguity range passed in and the current layer's contiguity, adjusts for framing, and passes this value to the next layer. The contiguity value delivered to the application layer (the MetaXyn terminus, in this case) is the target range the application must observe in generating data to ensure the IO data is delivered intact to the receiving application endpoint.

By collecting attributes such as contiguity from a layer hierarchy, computing a valid configuration or searching for an optimal one among a set of possible configurations, and propagating the results back to the individual layers, MetaXyn minimizes tensions arising from incomplete layer isolation. Since the information shared is derived from the *abstract* syntax descriptions of the IO data imported and exported by each layer (i.e., the layer interface definitions specified using Xyn), the sharing enforces the isolation between layer implementations, and hence does not violate the integrity of the IO system architecture.

In contrast to GenVoca [Batory and O'Malley, 1992] or Aspect Oriented Programming [Kiczales et al., 1997] optimization achieved by specializing (e.g., type parameterization), adding, or eliding functional layers, MetaXyn optimizes on the data access path to minimize access overhead for a given (i.e., assumed fixed) layer configuration. MetaXyn is closely related to Click's dataflow analysis to determine where to insert "alignment" (i.e., copy) elements; however MetaXyn is more general in that it derives the information from each layer programatically, rather than using a table of per-element alignment requirements. Additionally, MetaXyn provides feedback to Xync to generate layers statically optimized for identified inter-layer constraints. MetaXyn shares with (or exploits) typical IO processing frameworks the ability to attach meta–data to a message to pass through the layer hierarchy — in MetaXyn's case, the meta–data relates to syntactic attributes, rather than the more common signaling payload (e.g., "bind" or "flush channel").

In summary, MetaXyn is not a specification language in which to express inter–layer relationships, but instead extends Xyn with a framework in which algorithms can be deployed to operate on selected per-layer attributes, and a meta–layer protocol to propagate results through the layer composite. Each MetaXyn managed attribute is paired with an algorithm to compute a cost function or evaluate a solution space for that attribute. An algorithm is invoked on a layer when the attribute–specific meta–data is delivered to that layer, or it is triggered by a local state change.

The idea for MetaXyn originated from the need to integrate the extensively developed and analyzed user–system boundary copy–avoidance techniques (e.g., virtual–memory page remapping) into the complete IO data processing path in order to expose the true characteristics of the copy–avoidance interface to its adjacent layers. This application of MetaXyn is discussed later in the chapter.

Note that this chapter describes the *design* of MetaXyn, rather than an implementation. Fully integrating MetaXyn's requirements and optimizations into an existing IO system requires reimplementing aspects of each IO data processing layer (e.g., redeploying using Xyn) and the encompassing software IO framework. Considering the scope of such an effort, and the dissertation's focus on intra–layer IO data access, an implementation of MetaXyn is left as a subject of future research. Instead this chapter demonstrates MetaXyn's utility and symbiotic relationship with Xync through detailed examples of how controlled sharing of syntactic attributes offers a straightforward approach to improving the robustness and performance of a layer composition.

The chapter is organized as follows. Section 6.1 introduces the inter–layer complexities involved in meeting the IO data syntactic *layout* constraints that are key to efficient intra–layer access mechanisms. The section continues with a description of how MetaXyn utilizes the intra–layer *contiguity* attribute to implement *Application Layer Framing* [Clark and Tennenhouse, 1990] and provide the contiguous IO data requisite to Xyn's statically optimized access bindings. Section 6.2 describes the second layout attribute, *alignment*, and how MetaXyn initially analyzes a *proposed* composition of layers to provide input to the per–layer source compilation, and subsequently analyzes a *deployed* layer composite to compute the optimal alignment at which IO data should be delivered to the composite. Section 6.3 describes the non–syntactic attributes along the IO data processing path that interact with copy avoidance optimizations, including the user–system virtual memory protection boundary. Section 6.4 shows how MetaXyn introduces a degree of type safety to the layer composition by exploiting the de facto relationships between adjacent layers contributing to some larger service. Finally, Section 6.5 concludes the chapter with a review of MetaXyn's design principles and its contributions.

Used as an example throughout the chapter, the network video receiver shown in Figure 6.1 serves to illustrate the various inter–layer algorithms implemented in MetaXyn. The IO data processing portion of the application is a conventional layered implementation; utilizing well–known components to process the IO data in its "wire syntax" format. While these layers were developed to work together, it is enlightening to examine their interactions and provide an explicit formulation of the costs associated with each layer's IO data access. Of particular interest is the application of MetaXyn's algorithms across the user–system boundary, which is a key factor in minimizing IO data access overhead due to copying.

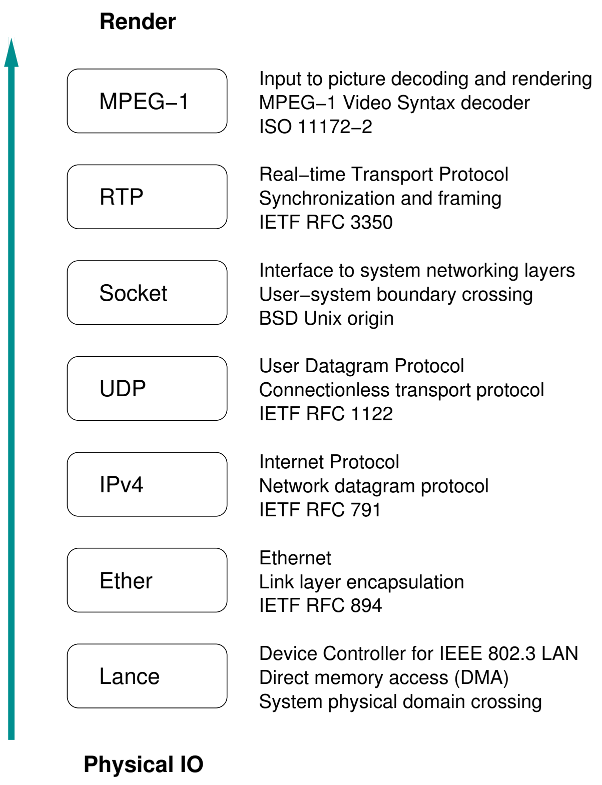
Render
MPEG−1
Input to picture decoding and rendering
MPEG−1 Video Syntax decoder
ISO 11172−2
RTP
Real−time Transport Protocol
Synchronization and framing
IETF RFC 3350
Socket
Interface to system networking layers
User−system boundary crossing
BSD Unix origin
UDP
User Datagram Protocol
Connectionless transport protocol
IETF RFC 1122
IPv4
Internet Protocol
Network datagram protocol
IETF RFC 791
Ether
Ethernet
Link layer encapsulation
IETF RFC 894
Lance
Device Controller for IEEE 802.3 LAN
Direct memory access (DMA)
System physical domain crossing
Physical IO

Figure 6.1: A network video receiver. The video data arrives at an Ethernet interface attached to the host's physical IO subsystem. The Lance Ethernet controller copies the data into the hosts main memory, where it is processed by the sequence of layers; each layer implementing a service coordinated with its remote peer. The final layer of the sequence converts the data from its wire (i.e., serialized) format into data structures defined in the video player's host programming language.

6.1 Layout

Xyn intra–layer accessors impose *layout* constraints on the underlying IO data, requiring each IO data field to meet *alignment* and *contiguity* constraints in order to be accessible to the binding code. As demonstrated in Section 5.3, effectively meeting these data layout criteria is mandatory for performant data access. In particular, making static (i.e., at compile time) assumptions about each IO data field's layout is critical to achieving the performance potential of a statically compiled language. While dynamically accommodating an unsatisfied layout constraint *per–field* is possible, it adds a significant overhead to the non–exceptional case, and is typically not even the most effective way to handle the exceptional case. Instead, a layer's layout constraints reflect the per–message *composite* of the per–field bit–access level static layout constraints. In turn, the layer's aggregate static constraints are imposed on the buffering layer, which must deliver IO data meeting the layout constraints; even copying the data into a conforming layout, if necessary. Hence an effective inter–layer binding facility satisfies per–layer layout constraints in a coordinated fashion for a composition of layers, minimizing copying and its impact on performance.

While satisfying layout constraints without copying is necessary for optimal performance, there are situations that require copying, such as incompatible constraints in adjacent layers or between locally optimized collections of layers composed at runtime. When a copy or copy–like operation must occur, the resulting layout should be optimized for subsequent accesses. A transformation of all or even most of the data can be used to effect a copy. For example, *secure sockets layer* (SSL) decryption of discontiguous segments delivered by a network layer could be used to deliver contiguous, aligned data to upper layers, at some additional overhead.[1] Thus an additional feature of an effective interlayer–binding facility is to recognize when a copy is needed, and to strategically place copies or exploit copy–like operations to minimize the total amount of data copying.

Xyn's interface to this inter–layer binding system is through *MetaXyn.* Xyn and MetaXyn share a symbiotic relationship: The Xyn compiler Xync provides characteristic attributes per syntax (i.e., layer), while MetaXyn gathers and evaluates attributes in aggregate, then publishes a set of attributes to instantiate the collection of layers for maximum efficiency. These meta–layer protocols coordinate inter–layer copy avoidance techniques to identify and eliminate unnecessary copies. The remainder of this section describes the *contiguity* layout attribute introduced above, and discusses how MetaXyn enables *Application Level Framing* (ALF) [Clark and Tennenhouse, 1990]. As described in the chapter introduction, ALF ensures per–layer messages are delivered intact, thus guaranteeing contiguous IO data required for efficient layer processing. Following this section, Section 6.2 discusses how MetaXyn assists in computing the static alignment value and ensuring IO data is delivered on that alignment.

Note that since the ALF and alignment attributes are per–application–flow, ALF

[1]Splitting an in–place update into distinct import and export buffers can add significant overhead when the CPU data cache has insufficient capacity or associativity.

deployment depends on a flow–oriented IO processing architecture at the application endpoints. While implementing such an architecture is beyond the scope of this dissertation, per–flow IO processing is not a significant departure from current layered implementations. In particular, layer attributes are currently maintained on a per–flow basis, for example, attached to the socket data structure in a typical network implementation. The key additional technique required to implement per–flow IO processing is *early demultiplexing* at the receiving application endpoint, which is described in [Druschel and Banga, 1996]. Thus, assuming such a flow–oriented IO processing implementation, ALF utilizes an attribute dynamically updated by a meta–layer protocol deployed along the vertical cut through the IO data processing layers.

6.1.1 Contiguity

The principle of application level framing (ALF) [Clark and Tennenhouse, 1990] is to observe the path *maximum transfer unit* (MTU) when packaging data for transmission, so that the destination application endpoint receives integral units of data that it can process independently; specifically, deal with out–of–order. The *path* MTU is defined as the minimum MTU among the processing layers between the origin application end–point and destination application end–point, including constraints along the network path between the end–points. The scope of an ALF computation is typically from the application's interaction with the presentation processing layer (e.g., constraining the length of an encoded MPEG *slice*, which is a sequence of contiguous macroblocks [2] from a picture's encoding), to the system's physical interface with the link layer device. A networking layer, such as IP in the example, can provide information about the network(s) between the application endpoints. The upper boundary of the scope is the result of the profound transformation of the IO data syntax; in particular, the size reduction due to encoding techniques used, which makes passing MTU information any further pointless.

While primarily intended to minimize latencies in data delivered to delay–sensitive applications such as interactive audio and video presentation, ALF also reduces IO data processing overhead by eliminating segmentation and reassembly overhead. Ideally, the

[2] A macroblock is a 16x16 pixel area of a picture. Macroblocks are ordered in rows; left to right, top to bottom.

application originating IO data observes ALF, but the technique is beneficial at any transmit layer that might perform segmentation. Furthermore, advantages of operating system IO data copy avoidance techniques such as "zero–copy" IO can be exploited all the way to (at least) the presentation layer, since each layer's IO data processing can operate directly on contiguous data residing in the system's IO data buffers (which have been mapped into the application's address space by some copy–avoidance scheme). It is interesting to note that the next generation of IP, IPv6, disallows network–layer segmentation, which indicates both the recognition of segmentation's drawbacks, and that the internetwork (path) MTU will become readily available at the communications endpoints.

6.1.2 MetaXyn Implementation

Implementing ALF requires a per–layer *maximum transfer unit* (MTU) attribute — more generally, a range is necessary since there may also be a lower bound on message size — which has the following characteristics:

- A layer's MTU is an attribute of its syntax, i.e., the MTU depends on all the message types traversing the layer along the vertical cut (and the syntax is the layer's interface to the cut);
- An MTU is specified only for a uni–directional flow since output and input data could traverse different network elements;
- The per–layer packet data unit (PDU) size is typically a range (*min..max*);
- A layer reports an MTU (range) that is the intersection of the MTU received from the layer below and its own PDU, minus any header or trailer (i.e., framing overhead);
- A layer's actual MTU value might not be immediately available; e.g., the IP layer must probe for inter–network path MTU, so the IP would initially report a value based on its default MTU value (derived from the syntax) and local information, such as the passed in link–layer MTU;
- The MTU value might be dynamic; e.g., due to a network routing change.

As an example, the minimum IPv4 message is the fixed 20 byte header, while the maximum is constrained by the 16–bit length field (i.e., 2^{16} bytes), leading to this Xyn specification:

```
  syntax IPv4Net { MX.MTU min=20 max=65536 ; }
```

Xync generates an MTU object similar to Xyn's semantic variables, with accessors for the min and max fields and initialized with the given default values. Additionly, Xync augments the layer's syntax (IO data interface) with optional meta–data fields so that MetaXyn can pass MTU data between layers (e.g., as a dedicated control message or piggybacked on an IO data message). When MetaXyn delivers MTU meta–data to the IPv4 layer, a flag indicates its availability and the ALF routine for the layer is invoked to compute the a new value for the IPv4 layer's MTU fields. Writing to either of the fields initiates a MetaXyn message to the next layer, repeating the process there. This method of initiating MetaXyn processing is also used by the IPv4 layer itself when it receives new path MTU information from its peer.

MetaXyn computes the layer composition's target ALF attribute by propagating an attribute value up through the layers, taking the intersection of the propagated value and the current layer's value, as described above. The result is a target range for application IO data generation. An empty intersection, or failure of the application to conform to the target range (e.g., an ALF–ignorant application), implies some service layer needs to accommodate by segmenting the transmitted data. Since this segmentation occurs without regard to IO data syntax boundaries, the segments must be reassembled at the corresponding layer at the receiving endpoint for delivery to the upper layers. Additionally, should the target range change for some layer during operation, the updated ALF attribute propagates upwards from that layer until either the application is reached or the lower layer's update does not result in an attribute change in some layer.

ALF integrates naturally with Xyn's intra–layer binding mechanisms. Xync computes and provides each layer with a set of *size* attributes, providing static minimum and maximum PDU sizes, that are necessary for MetaXyn's ALF computations. In turn, ALF guarantees Xyn's bit–access layer contiguous data, which is necessary for Xyn's efficient binding implementation. Along with improved efficiencies, a symbiosis in message validation arises from Xyn's (dynamic) per–message size attribute combined with the ALF processing, which would facilitate out–of–range message sizes to be automatically detected and discarded. A hook into the mechanism would allow reporting such an exception locally, or even to the sender, should the layer semantics require it.

A future version of Xync might be able automatically compute the initial values of the MTU attribute by analyzing information provided by or implicit in the Xyn specification, rather than the current design's approach of using an explicitly declared MTU attribute. In particular, a future version of Blex will utilize the per–field range attribute, which would allow IPv4's MTU attribute to be inferred by the range of values assigned to the IPv4 syntax's length field.

6.2 Alignment

IO data alignment complements contiguity, and is the other key IO data layout attribute on which Xyn's effective IO data binding mechanisms depend. A host buffer alignment is defined as an d–byte (positive, i.e., trailing) displacement from some m–byte boundary, and written m/d. The value for m is constrained to be the size of a host integral type (e.g., the C language's `char`, `short`, `int`, or `long` types): an m–byte alignment boundary is defined wherever $address \bmod m = 0$. The displacement d is constrained to $0 \leq d < m$.

In contrast to Blex defined syntax fields (i.e., Xyn's terminals), bit–granularity alignment is not defined for buffers, since host addressing is at the granularity of a byte. This requirement implies a coupling between the syntax types and host architecture — namely, that field size is related to host integral size. In particular, throughout existing syntax specifications there is an assumption of an 8–bit byte; i.e., a byte is the same size as an octet.

Alignment constraints differ for sequential access syntax processing (SEQ) and random access syntax processing (RA). The IO data access mechanism developed for SEQ processing must accommodate arbitrary IO data alignment, due to aggregates and sequences of variable length encoded Blex fields. Bit–access and buffering layers specialized for SEQ syntaxes provide efficient access for these constructs by mandating sequential (i.e., unidirectional) field access, described in Section 4.2. In contrast to the minimal SEQ accessor alignment requirements, alignment is a key attribute of an RA syntax implementation. Integral alignment is mandatory for most architectures; i.e., access to a one (two, four) byte element must fall on a one (two, four) byte address boundary.[3] Based on the assumed IO data displacement (i.e., the alignment of the first field of the *Xynion* or *SubXynion*), a field's bit–offset within its resident frame(s) must be fixed at layer compilation in order to reduce access cost to a minimal sequence of load, store, shift and mask instructions to access the desired bit–field. The remainder of this section describes how this is accomplished for RA syntaxes.

6.2.1 Overview

Minimizing the total overhead resulting from layers recovering from misaligned data requires sharing per–layer alignment constraints among layers. Each layer reports an optimal alignment and user offset (e.g., header size), and a cost when the alignment constraint is not met. In some cases the constraints might be fixed; for example, the cost of not meeting some hardware constraint is infinite. The MetaXyn meta–layer protocol collects cost and layout information, computes an optimal alignment at which the originating layer should deliver data, and provides the target alignments and offsets to the layers. When the algorithm is applied to a static configuration of layers (i.e., offline, or prior to deployment), the constraint information might be used to (re)compile one or more layers to operate more efficiently at the negotiated alignment. This technique is useful up to and including the MetaXyn deployment phase (assuming the compilation environment is available on the target host), and accommodates platform specific optimizations such as (sub)layers encapsulating encryption or checksum computing enabled hardware; however, compilation times preclude its use during runtime initialization or reconfiguration.[4]

Post–compilation analysis is used during the deployment and runtime initialization of a composition of layers. Since each layer has been statically configured and compiled for a specific alignment, MetaXyn's role is to compute the optimal alignment at which IO data should be delivered to the layer composite. In this case, a layer satisfied by an alignment will

[3]The Intel x86 architecture is a notable exception: the ISA's variable length instructions demand efficient access to memory at arbitrary byte alignments.

[4]An alternative approach would be a library for each layer consisting of a layer instance compiled for each possible alignment. In practice, this is has so far proven unnecessary since 1) collections of layers belonging to a suite will be compiled and deployed together, and 2) these composite layers (i.e., suite) are designed to deployed into a particular position of a conventional IO processing architecture.

report the static cost for the alignment, otherwise the layer reports the static cost plus the cost of copying sufficient IO data to meet the static alignment constraint. The remainder of the section describes in detail the per–layer access metrics, how the user–system protection boundary is integrated into the IO data path, and the inter–layer metrics.

6.2.2 Intra–layer Metrics

MetaXyn requires intra–layer alignment metrics in order to compute the optimal inter–layer alignment. Xync provides these metrics in the form of a list of per–displacement cost functions. The length of the list implies the layer's alignment modulus, which is the smallest host integral size containing the largest syntax (Blex) field size. Thus the list length will be a power of two and typically limited to no more than eight entries (i.e., the size in bytes of a long integer on a 64–bit architecture). The list index is 0–based, and for each list index d, Xync computes an access cost function for an alignment displaced d bytes from the modulus. The per–displacement cost function takes a length argument; typically this would be the length from the contiguity metric described in Section 6.1.1. In some cases, a layer only access a prefix of the IO data, such as the IPv4 header, and the length argument is ignored. The various cases are described next, in increasing order of complexity.

Static (Offline) Analysis

Evaluating the per–displacement "fit" for a static syntax (i.e., no dynamic elements such as *Choice* or *Sequence*) is a straightforward summation over the syntax's fields:

- Base per–field cost is 1;
- +1 for each frame boundary crossed; i.e., the integral type of size n (modulus), above;
- +1 for each (sub)frame requiring shift–mask to extract or insert the field in addition to integral access.

and is denoted as $Cost_{prefix}$. This value approximates the relative costs of layer access, by increasing the total cost for each overhead inducing operation.

The cost for a syntax consisting of a static prefix followed by a *Fixed Sequence* (i.e., static–type, such as a sequence of a Blex type), can be computed similarly. In this case the cost is computed for the first sequence field, and then multiplied by the length of the *Fixed Sequence*. The cost of the access is estimated as:

$$Cost_{prefix} + Cost_{element}(Length_{ALF} - Length_{prefix})/Size_{element}$$

where $Length_{ALF}$ is from the contiguity attribute computation, and is passed as a parameter to the cost function. Recall that the *Fixed Sequence* specification constrains the element size to an integer multiple of the initial element's alignment modulus; so that all elements will be aligned identically to the first. In the case of multiple Xyn *View* alternatives, the cost of the *View* is computed for the alternative appearing first in the specification (as is the case with other *View* attributes). Note that multi–message syntaxes, such as found in RPC, are described by the *Choice* construct — a *View* is a simpler construct.

Xyn's dynamic constructs, *Choice* and *Sequence*, obfuscate the cost computation due to the type variations possible for a syntax instance. While it is possible to expand an AST with all *Choice* and *Sequence* possibilities (a message should always be of finite length) in order to compute worst–case cost, or perhaps an average computed across the expanded AST branches, it is unlikely the potential computation and state overhead could be justified. Instead the current MetaXyn design uses a simple heuristic. RA syntaxes with dynamic parts typically have a common (static) prefix followed by the dynamic part; IP being an example. Assuming the syntax was designed so the dynamic part's optimal alignment is satisfied by optimally aligning the prefix, compute the layer's cost as:

$$Cost_{prefix} + Cost_{prefix}(Length_{ALF}/Length_{prefix})$$

Finally, to override an undesirable result (e.g., through some specialized knowledge), or in the absence of a static prefix, MetaXyn allows an *align* directive in the Xyn syntax to specify frame alignment information. The *align* directive is positioned at the start of *Xynion* and *SubXynion* construct specifications, corresponding to the point in the syntax where buffering decisions (e.g., misalignment recovery) are made during runtime, and has the syntax:

$$\text{align } n_0, n_1, \ldots, n_{m-1}$$

where m is the alignment modulus and n_d is the user–specified cost analysis at displacement d from an m–byte alignment boundary. Each cost analysis is syntactically similar to a Xyn

Semantic Variable (i.e., curly braces surrounding a simple computation), and can refer to the message length parameter passed in at MetaXyn analysis time. The *align* directive is translated into an array of functions indexed by displacement and identical in structure to those generated from the heuristics described above.

Deployment Analysis

Compared to the static analysis, above, the metrics necessary for deployment analysis are simple. The layer must provide an *offset* between the IO data on the transfer syntax side of the layer, and the provider syntax on the layer's opposite side. The sum of the displacement of the IO data on the import side of the layer and the layer's offset determines the displacement of the IO data produced on the export side of the layer (and delivered to the next layer along the flow). For example, the IP header (without options) is 20 bytes, hence IO data delivered to IP by the network layer below, will be provided to IP user offset by 20 bytes. Hence, for alignment moduli up to and including 2^2, the IP offset will not affect the next layer's IO data displacement.

The other per–layer deployment metric is the per–displacement *access overhead* vector, which is the cost of copying misaligned accessed IO data to the layer's statically compiled alignment. For an alignment having modulus m, for each displacement $d \mid 0 \leq d < m$, other than the displacement for which the layer was compiled, the access overhead is simply the number of bytes accessed. In some cases, such as IP, this is a fixed value; however, in general, it is the contiguity (i.e., IO data length) value from the Section 6.1.1, above. The description of inter–layer metrics, below, includes details on the use of the per–layer deployment metrics.

6.2.3 Inter–layer Metrics

While contiguity is an end–to–end attribute and computed along the entire vertical cut from application to link–layer interface, an alignment attribute is local to a host and its scope might be limited to a contiguous subset of the IO data processing layers. The scope of a MetaXyn alignment computation is called an alignment *coupling*. Like contiguity, alignment attribute endpoints occur where there is a profound transformation of syntax (distinct

MPEG–1 (ISO-11172) Video layer

- **provides** N/A (profound IO data transformation)
- **requires** 1/0 (packed stream, variable length coding, uses SEQ access)

RTP Header processing, optional CSRC identifiers

- **provides** received, offset by $12 + 4 * n, 0 \leq n \leq 15$ bytes
- **requires** 4/0

Socket (system boundary, pseudo–layer) System virtual–boundary crossing

- **provides** received
- **requires** 4/0 (reflected from RTP)

UDP Header processing, cksum of full IO data, demux to port

- **provides** received, offset by 8 bytes
- **requires** 2/0

IPv4 Header processing, IP optional header fields, demux to protocol

- **provides** received, offset by $20 + 4 * n, 0 \leq n \leq 10$ bytes
- **requires** 4/0

Ether (RFC894) Link layer processing (header, demux)

- **provides** received, offset 14 bytes
- **requires** 2/0

Lance (DMA) System physical boundary crossing

- **provides** 1/0 (more restrictive might improve burst performance)
- **requires** N/A (hardware, physical boundary)

Table 6.1: Per–layer alignment requirements for the network video decoder example. Input IO data flows from bottom to top; i.e., IO data is imported from the network side of a layer and exported to the application side. An alignment value m/d expresses a displacement of d bytes from an alignment modulus of m bytes, and is the alignment requirement for minimal access overhead within that layer. The per–layer *offset* provided is required to compute the alignment of the IO data delivered by the layer.

import and export syntaxes), such as that resulting from a presentation processing layer implementing compression, or at the system's physical layer interface with an IO device. Within these maximal endpoints, one or more additional alignment coupling boundaries and associated scopes might arise due to a copy or copy–like operation, such as a mandatory copy due to a persistent reference, or a system boundary crossing not eliminated by a copy avoidance technique.

The inter–layer metrics are deduced by analyzing the relationships among the intra–layer metrics of the sequence of composed layers. Each IO processing layer imports and exports data. The import phase imposes alignment constraints on passed–in data that are necessary for layer processing, while the export phase imposes constraints on the data it emits to the next layer. From an IO processing configuration, MetaXyn first identifies *coupling* boundaries; i.e., IO processing layers spanning a physical boundary, that implement a profound transformation of the IO data layout (e.g., compression), or, weaker, implement a copy, or copy–like transformation that does not fundamentally alter the data–layout (e.g., encryption). For each coupling, MetaXyn computes the alignment IO data should be delivered to the coupling to minimize copy overhead.

The network video decoder's layer alignment requirements are summarized in Table 6.1. The per–layer entry *requires* specifies the alignment of the delivered IO data to avoid (or at least minimize) copying at import for this layer's processing. The *provides* entry describes the alignment of user data at export (the downstream side). In some cases, such as when the user (export) data container is decoupled from the import data, the provided alignment is an absolute value dependent on the exported datatype. In other cases, the import data is passed through to the export, hence the export data is aligned relative to the import data and a layer header results in an offset. The offset value is important in computing optimal alignment, since the alignment displacement of some layer i depends on the sum of the (header) offsets of all layers below layer i.

An example of these interactions occurs in the decoder example. Lance needs to start the DMA two bytes ahead of a four byte boundary to accommodate Ether's 14 byte header and keep Ether's user (IP) aligned, as shown in Figure 6.2). In this case, the origin of the IO data to the coupling should deliver the data at an alignment (address) divisible

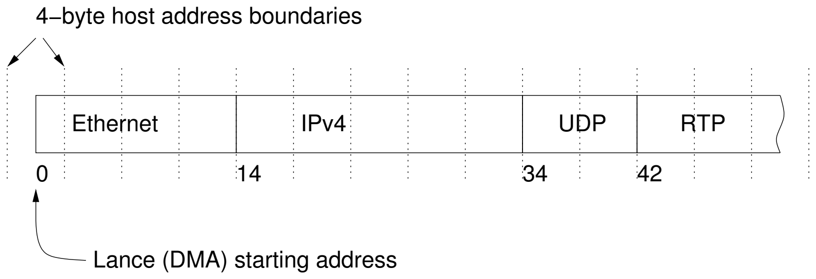

Figure 6.2: Displacement example. In order to align IPv4's header, Lance (DMA) needs to displace the IO data start two bytes from some four–byte boundary.

by two, but not by four; that is, displaced two bytes from a four byte boundary.

Alignment Cost Function

Given the per–displacement cost function and an offset value for each layer, as described above, computing the cost function for the coupling of layers requires a straightforward summation. First compute the aggregate offset for each layer.

$$Offset_i = \begin{cases} 0 & \text{if } i = 1 \\ Offset_{i-1} + offset_{i-1} & \text{otherwise} \end{cases}$$

where, $offset_i$ is the offset of layer i. Then, the aggregate cost function for the coupling, evaluated for each feasible displacement $d \mid 0 \le d < Modulus$, is

$$AggregateCost_d = \sum_{i=1}^{Layers} Cost_i((d + Offset_i) \bmod Modulus_i, Length_{ALF_i})$$

where $Modulus_i$ is the alignment modulus for layer i and $Modulus$ is the largest of these, $Layers$ is the number of layers, and $Cost_i()$ is the intra–layer cost function evaluated for a particular displacement and length — the length is from the ALF–based contiguity computation described in Section 6.1.1. The fundamental concept underlying the algorithm is that at a given initial displacement of the IO data, the IO data displacement a particular layer sees, which is the index into that layer's cost function, depends on the prefix consumed by the layers below. Note that the per–layer cost function could be from either the static analysis or dynamic analysis of the Intra–layer metrics.

While the computational complexity of the algorithm is high — $O(mn)$, where m is the largest alignment modulus and n is the number of layers — in practice m is typically 4 and n will be less than 10. Furthermore, an *AggregateCost* vector computed for a layer composite can be reused directly for an identical configuration, or used as the initial value for an configuration extended by adding layers either above or below.

Example

For the video decoder example, the cost is first computed for the layer coupling between the physical system boundary at *Lance DMA* and the virtual system boundary at *Socket*. The access computation is dominated by *UDP's* checksum of the entire IO data. Based on *UDP's* offset, the IO data should be delivered on a two–byte alignment boundary — i.e., a 2/0 alignment. *IPv4* offers an advantage when IO data is delivered to it on a four–byte alignment boundary. Due to *Ether's* introducing a 14–byte offset, delivering the IO data to the layer composite displaced two bytes from a four–byte boundary, i.e., a 4/2 alignment, will optimally satisfy all the layers. In fact, this is a well known requirement for IP using Ethernet for a network layer.

When *RTP* and *MPEG Video* layers are added to the layer composite across the virtual system boundary crossed by *Socket*, MetaXyn needs to determine if *Socket's* copy–avoidance mechanism can be used effectively. Interestingly, while *RTP's* four–byte optimal alignment is satisfied by the layout offered by the layer composite below *Socket*, due to the relatively small *RTP* header size the deciding factor in favor of the copy avoidance technique is *MPEG's* imposing no alignment constraints in accessing the entire IO data delivered to it.

MetaXyn Deployment

MetaXyn's alignment analysis is valuable through a configuration's initialization phase. To optimize a static (i.e., one proposed at compile–time) configuration, MetaXyn examines a configuration graph, Xync generated source tables, and recommends per–layer alignment displacements for layer compilation. If the compilation environment is available at configuration deployment on a host, MetaXyn could again be run, optimizing for the characteristics of the particular host. Finally, during configuration runtime initialization, MetaXyn examines the configuration graph, queries each layer for its compiled alignment, and computes the alignment to request IO data be delivered.

6.3 Non–Syntactic Considerations

The previous section described inputs to MetaXyn's alignment computation that are based on per–layer syntactic restrictions on layout. In addition to syntactic constraints, non–syntactic factors have a role in a comprehensive inter–layer copy–avoidance scheme. This section first reviews how mandatory copies and copy–like operations enable possible MetaXyn optimizations. This introduction is followed by an example describing how the user–system virtual memory protection boundary is integrated into MetaXyn's optimizations, in particular, how MetaXyn determines whether to use a virtual–memory remapping based copy avoidance scheme, or exploit the copy to mediate between layer couplings with different IO data layout expectations. The section concludes with a description of how a layer's persistent reference to IO data processed and passed downstream interacts with copy–avoidance mechanisms.

6.3.1 Mandatory Copies and Copy–like Operations

In certain cases it might be feasible to optimally locate the mandatory copy due to a persistent reference (somewhere between the layer holding the reference and the layer modifying the data), when it would suffice to replace a copy operation inserted only to satisfy an inter–layer alignment conflict. Additionally, an in–place transformation such as a block encryption or decryption or even a scan (i.e., a read–only traversal) over some IO data to compute a checksum could also be exploited as an alignment coupling boundary. Note that these latter techniques are not available in the current Xync intra–layer binding implementation — the access and buffering mechanisms do not support inserting a copy operation in the case of a scan, or directing the result of a computation to an alternative buffer in the case of an update. Assuming the need to exploit one of theses operations could be identified prior to runtime initialization (as long as the compilation environment is available), such a technique could likely be integrated into the Xync generated bindings without interfering with the extensive compiler optimizations currently achieved for the accesses.

6.3.2 Protection Boundary

The user–system protection boundary crossing poses a constraint different from those of the syntax–binding layers described in layout section, above. Its responsibility is to deliver its provider's data intact to its user, without consideration of the values (content) of the data. In a conventional system this layer is implemented using a copy across the protection domain boundary, and the *socket* or *file* programming interface exploits this copy to accommodate differences between the provider and user layouts. Through virtual–memory page–table manipulations, the copy and its resulting overhead can be largely eliminated; however, only if the following criteria are met:

- The layouts of provider and user must be compatible; otherwise, the conventional copy can be exploited to accommodate the layout requirements of the destination;
- The virtual–memory page–sized alignment and granularity constraints of the protection boundary must be accommodated;
- For any physical memory page passed into a process, the content that did not originate from that process or is not destined for that process must be cleared (e.g., filled with zeros);

These criteria are common to, and described with, any of the user–system boundary copy avoidance schemes in the literature, such as those surveyed in Section 1.2.2 of the Introduction.

Layout Constraints

To facilitate meeting the first constraint (i.e., compatible layout), MetaXyn operates in one of two modes. In the first mode, the boundary crossing layer presents itself as a VM page–aligned, page–granularity "do nothing" IO data processing element. This mode is useful for IO data channels that are integrated with the VM system, such as memory–mapped or raw–mode file–system operations, and operates with optimal efficiency due to the absence of copying.[5]

In MetaXyn's second mode, the boundary crossing layer must support arbitrary (i.e., single–byte) alignment of IO data in order to accommodate the irregularly structured IO data typical of networked communications. At the same time, the boundary crossing layer must interact with adjacent layers to ensure an embedded copy avoidance mechanism is used to maximum advantage. MetaXyn presents the boundary crossing layer differently depending on the configuration phase. When the boundary crossing layer is at the top or bottom of a configuration — e.g., when a subsystem such as IP is being *statically* configured for compilation — the boundary layer presents itself as a copy operation that accepts or provides arbitrarily aligned IO data. Accepting arbitrarily aligned IO data means the layer can access IO data falling on any alignment boundary, while providing arbitrarily aligned data means the layer can provide IO data to any requested alignment. Using this representation for a coupling's endpoint, the coupling is locally optimized (i.e., among its constituent layers) at compile–time.

When the boundary crossing layer is an intermediate layer of a configuration — e.g., when an IO channel is being *deployed* or *initialized* for the network video receiver — MetaXyn's objective is to utilize the capabilities of the boundary crossing layer to mediate between the layer couplings on its opposite sides, seeking to globally minimize access overheads. If the upstream export alignment satisfies the downstream import alignment (and the other copy avoidance constraints are satisfied), then the IO data resident memory pages can be mapped into the destination address space. Otherwise, the layer's (presumably highly) optimized copy mechanism can be used to deliver the IO data appropriately aligned. The details of the boundary crossing layer intra–layer metric are at the end of this section; the inter–layer metric computation is given in the next section.

Virtual–memory Page Constraints

If the second constraint (i.e., VM page alignment and granularity) is not satisfied by a single IO data message, then it might be effectively satisfied by an aggregate of messages

[5]Another approach would be to extend the range of candidate IO data alignment moduli beyond the current integral type sizes (e.g., 2^0–2^3 bytes) to include the VM page alignment boundary; typically 2^{12} or 2^{13} bytes.

waiting to cross the user–system boundary; which would often be the case in a loaded, multi–tasking system. Effective satisfaction requires the aggregate fills one or more pages densely enough to reduce the cost of satisfying the third constraint — inter–domain privacy — below the cost of simply copying the IO data across the boundary. Satisfying the VM page constraints requires a significant (though not original) departure from conventional user–system boundary crossing, since it violates the conventional (i.e., synchronous IO) behavior of a buffer being immediately available following a return of control from the boundary crossing implementation (e.g., socket read or write). Instead, by deploying a *ring–queue* on the import (i.e., upstream) side of the boundary crossing, buffer aggregates can accumulate while awaiting a request for more IO data from the destination side. Hence, at minimum, the user (i.e., at the import or upstream side) of the boundary crossing layer needs to adapt to an asynchronous IO interface. To more fully exploit the copy–avoidance boundary crossing mechanism the user–side application should be modified to request an IO buffer from the boundary crossing layer; this allows the layer to return adjacent buffers in response to a series of allocation requests.

Privacy Constraints

The final constraint (i.e., hiding from a user process any data on a physical page to be mapped from the system into the user process address space that was not originated by, or is not destined for, that user process) requires zeroing any unused space on a physical page that was previously mapped into another process (including an unrelated purpose in the system virtual address space). At the expense of reducing overall availability of memory, the IO subsystem could maintain a pool of physical pages, each marked with an affinity for the process it was most recently mapped into. [Brustoloni and Steenkiste, 1996] presents a extensive analysis of such techniques.

Access Metric

In order to seamlessly fit into MetaXyn's inter–layer optimizations, described above in Section 6.2.3, the boundary crossing layer would have to provide a per alignment–displacement access cost metric, similar to that provided by each IO data processing layer. In fact, the per alignment–displacement metric is not applicable to the boundary crossing layer: since the layer does not interact with the IO data content, it has no static preference for any particular displacement. Instead, the boundary crossing layer can be thought of as having a variable *offset*; however, with some additional overhead when the offset is not 0.

Recall that the inter–layer metric computation determines an optimal alignment at which to deliver IO data to a layer composite. Since the boundary crossing layer is oblivious to the alignment of IO data received, the inter–layer optimization needs to determine when an alignment mismatch should be dealt with by the user–system boundary crossing copy, or by an IO data processing layer (so the VM page remapping is used to cross the user–system boundary). In order to correspond to the intra–layer deployment metrics, the cost function for copying is simply the number of bytes to be copied; however, this value might need to be scaled in case the cost of crossing the protection boundary is different from a copy within a domain.

Computing the cost of the mapping for the $offset = 0$ case is more complex. First, in order for the costs to be comparable, the absolute cost of the page–table manipulations must be expressed in terms of the cost to copy some number of bytes. Second, the typical overhead per remapped VM page necessary to satisfy the privacy constraint — for example, clearing "unused" space between IO data segments — must be added to the base remapping cost; presumably this can be acquired from embedded counters. Third, the *per–message* overhead of the remapping is computed by multiplying the overhead from the previous steps by a factor resulting from dividing the size of the IO data per message (i.e., the contiguity attribute) by the average number of IO data bytes on a remapped VM page. Finally, since there will be instances when the page–remapping is not, or cannot be applied, the final cost is

$$\rho \times Cost_{copy} + (1 - \rho) \times Cost_{remap}$$

ρ is the probability a copy is used, and is also computed from accumulated counter data. This computation can be incorporated into MetaXyn's inter–layer optimization at the point the alignment costs of layer couplings have been computed.

At runtime initialization by MetaXyn's runtime protocol, the boundary crossing layer records the alignment and contiguity constraints of its adjacent destination layer. When servicing a request, the layer compares the constraints with the IO data it is to deliver. If the IO data is compatible with the destination's constraints, and the other boundary–crossing copy–avoidance constraints are met, then the IO data can be delivered to its destination using a VM page remapping. Otherwise, the boundary crossing layer copies the IO data to a layout conforming to its downstream neighbor's constraints.

6.3.3 Access Mode and Persistent References

A non–syntactic optimization to which MetaXyn can contribute is the relationship between an upstream layer maintaining a persistent reference to some IO data that is to be modified by a downstream layer. Consider network layer encryption. An example of this situation occurs within the IP suite. First, consider the UDP transport layer, which provides a best–effort service and does not retain IO data awaiting an receipt for its delivery. Thus, assuming no other layer above UDP is maintaining a reference to the IO data, UDP's data can be encrypted in–place, reducing the pressure on the CPU cache.

Conversely, after the TCP layer transmits a message — containing a sequence of IO data from its associated socket buffer — TCP must retain a reference to this data until it receives and acknowledgment that the data sequence has be received intact. If the IP layer is encrypting this IO data flow, then the encrypted data must be written to a separate buffer. Otherwise, if the encryption is performed in–place, the data in the socket buffer is corrupted. Currently, Xyn accessors at the *Xynion* (i.e., message) level are declared to operate in one of three access modes: export, import, or update (i.e., import, then modify in–place). By adding an additional *persistent reference* attribute to this set, MetaXyn could compute an optimal buffering strategy that accommodates persistent references. This would imply a persistent reference on a layer (i.e., horizontal) peer basis, rather than one removed after delivery within the same coupling.

6.4 Layer Type

As discussed in Section 1.2 of the Introduction, conventional layered IO implementations allow substantial flexibility in layer compositions; relying on the implementor or deployer of the system to ensure adjacent layers are semantically and syntactically compatible. Since the service/semantic layers are bound through exchanged IO data, one objective of MetaXyn's inter–layer binding facility is detecting invalid compositions by comparing the IO data syntaxes at the interface of adjacent layers. Towards this end, a layer's *type* is defined as the IO data syntax accepted (on import) or produced (on export) at the layer's access point(s).[6] Type correct binding requires that an upper layer's transfer syntax and the adjacent lower layer's provider syntax be compatible; meaning the consuming layer's syntax must contain (in terms of the language accepted) the producing layer's syntax.

6.4.1 Type–safe Layer Binding

While evaluating syntaxes (languages) for the compatibility property in general is challenging[7], in reality, an IO data processing system is far from a collection of arbitrary syntaxes. Instead, within the overall hierarchy of more specialized layers building on more general ones, sub–hierarchies of related layers implement a service specialized by the particular layers chosen. The service provided dictates where the sub–hierarchy fits in the overall hierarchy, and the characteristics of its provider and transfer interfaces. Recognizing these structural constraints facilitates finding a feasible solution to interface compatibility validation. In fact, two cases typically occur:

- Layers are elements comprising single protocol layer (or protocol suite consisting of a collection of related protocols, such as TCP/IP), and the elements exchange specialized information (such as flow state) as an attribute prefix per–message, hence compatibility of adjacent layers can be verified by testing for a common syntax *label* assigned to the prefix by the implementation;

- A layer providing an interface to a service abstraction (e.g., the composite of layers of the first item), exchanges opaque (i.e., untyped) data with its users, as an artifact of layer isolation. Typically, at this granularity, the service would be well–known, an

[6]Note that in this context, type is purely syntactic. In general, expressing semantic attributes of a layer, and automatically evaluating the semantic properties of a composition of layers, are complex and are outside the scope of this dissertation. While syntactic typing is less powerful, it is much more straightforward, while still offering tangible benefits.

[7]IO data syntax compatibility is different from equivalence — the latter could be tested for by comparing the abstract syntax trees (created when a syntax is processed by Xync). Syntax compatibility is less strict, since an opaque (i.e., untyped) octet sequence in one syntax is defined as being compatible with any type in the compared syntax.

interface could be tagged with its functional name, such as the MIME type tags used by the *gstreamer* Multimedia Framework [Walthinsen, 1999].

An example of the first case is the *pseudo*–IP syntax that is the local user syntax of the IP layer and the local transfer syntax of the IP suite's transport syntaxes. Note that pseudo–IP is a superset of what goes "on the wire." Specifically, it is an encapsulation of the transport layer data with some IP layer elements unique to the TCP/IP session to which the IO data belongs. An example of the second case is the use of a MIME type such as *video/MPV* to tag an MPEG–1 video stream (specialized syntax for use without the usual encapsulating MPEG system layer syntax) as an RTP payload type (i.e., user syntax) and as a transfer syntax tag for an MPEG video codec parser or generator operating on the syntax.

6.4.2 MetaXyn Implementation

A service of a hierarchical IO processing architecture, operating on an IO data flow (i.e., along the vertical cut through the service layer), typically processes both input and output flows (i.e., bi–directional data flows; even if one direction is entirely control information). Hence a service's interfaces can be characterized by a type attribute table, with transfer versus provider interface along one dimension, and input flow versus output flow along the other. Each access point of the layer either imports or exports IO data typed according its corresponding table element. Such a service is typically composed from a collection of single–function elements (component layers), each operating exclusively on either the input or output flow. Internally, semantic operation(s) tie the flows together, as required.

The Xyn specification language top–level component *Syntax* describes an interface, so the corresponding label is used to tag the layer access point. The label *opaque* is reserved for an access point that imports or exports purely opaque data. Through interface type tags embedded in the compiled layer processing code, MetaXyn can validate a configuration graph by testing for matching syntaxes, and report the result. Typically, this test would occur in the initial MetaXyn pass for either a static validation (e.g., for a composite layer), or during deployment of an entire hierarchy. Validating the type–correctness of the composition should occur prior to any other processing performed by MetaXyn.

A component layer might implement a syntactic transformation, say transforming the IO data syntax between provider and transfer syntax, or some intermediate (internal) representation. For example, Click's *CheckIPHeader* element of Section 5.2 imports the opaque link–layer IO data and exports messages containing validated IPv4 network syntax. Subsequent Click IP router layers operate on the IPv4 network syntax, performing (semantic) transformations on the syntax field values.

In summary, the objective of type–safe inter–layer binding is twofold. First, in the case of non–opaque layer interface types, it ensures type–safe layer–layer binding, thus improving the validity of a layer composition. Second, even for opaque data interfaces, promoting the use of message syntax to exchange flow attributes or synopsis makes layer–layer interface (state sharing) explicit and more portable.

6.5 Contributions

Section 1.2 of the Introduction described how current layered IO architectures and implementations are the result of applying the fundamental software engineering principle of *separation of concerns*. At the same time, optimizations necessary to achieve effective IO processing require sharing information between the layers. The resulting challenge is how to break the layer encapsulation in a controlled manner, such that the integrity of the layered IO system architecture is maintained.

Observing the inter–layer architecture outlined in Section 2.2, this chapter showed how several intra–layer attributes exposed by Xync could be analyzed in aggregate by MetaXyn to optimize a composition of layers. In particular, the chapter described:

- Algorithms computing the optimal layout of IO data to minimize access overhead for the collection of layers comprising an application's IO channel;
- How to integrate non–syntactic layers or mechanisms into MetaXyn's optimizations, including exploiting copies such crossing the user–system protection boundaries;
- A limited form of inter–layer type checking based on comparing the syntax labels of adjacent layers.

At the same time, the layer encapsulation is minimally violated, since the attributes exported by Xync are derived from the abstract syntax specification of the layer, rather than through exposed internal implementation details.

While MetaXyn remains to be implemented, the chapter presented the design of MetaXyn in detail, including exposing several possible tradeoffs. Ultimately, MetaXyn needs to be incorporated into a larger IO subsystem architecture and design framework — in particular, one that spans from the application to the physical IO interface of the system.

Chapter 7

Conclusions

The continuing proliferation of diverse, IO–centric applications — including video conferencing, remote medical imaging, scientific grid computing, geographic information system (GIS) remote sensing, and WWW–based electronic commerce and information dissemination — requires the ongoing development of a wide range communication services and their intrinsic IO data processing facilities. One aspect shared by these services is the need to bind the semantic IO data processing operations to the elements of complex and externally defined IO data syntaxes. This chapter first summarizes the dissertation's solution of the problem of effective IO data access, and concludes with an exploration of future research directions.

7.1 Summary of Contributions

This dissertation designed the Xyn language to specify on–the–wire IO data syntaxes, and developed the Xyn compiler (Xync) facility to translate Xyn specifications into effective IO data access bindings in the C++ language. In the course of this work, the dissertation introduced a key abstraction boundary between the mechanisms of IO data syntax *navigation* and *access*, and program semantics or policies associated with the IO data *values*. By generalizing the technique of automated binding generation beyond the familiar RPC/RMI paradigms, the Xyn language further improves the typical programming practice by translating an IO data syntax specification directly into binding code. The dissertation described the experimental evaluation of Xyn bindings in the Click Modular Router, and while the performance of the generated bindings could not match that of the original C language style bindings, meeting this challenge appears to be feasible with additional work. A collection of micro–benchmarks developed to aid in identifying the sources of inefficiencies will be reused to evaluate new optimization strategies.

To augment this effective intra–layer IO data binding facility, the dissertation contributes a detailed design of an inter–layer facility that exploits per–layer transfer syntax information to improve the type–safety of layer composition and to optimize IO data layout for minimal overall access overhead. This controlled breaking of the layer encapsulation is accomplished using the abstract specification of a layer; without exposing details of the layer's implementation. A key contribution of the inter–layer facility is extending IO data copy avoidance techniques traditionally constrained to the operating system environment into the presentation and application layers. By introducing an improved discipline in the upper layer implementation, the efficiencies achieved in optimizing the system layers can be more fully exploited.

There are some fundamental insights that can be drawn from this work and applied to any DSL for IO data binding.

- *Syntax directed binding* The elements of the declarative syntax imply a mapping to target constructs in the host programming language domain, allowing the IO processing program (layer) implementor to express operations on the underlying IO data directly (i.e., using the syntax element labels);

- *Syntactic and lexical decomposition* The conventional approach to language processing is applicable to IO data syntax processing: a flexible, dynamic navigation component (Xyn) coupled with a statically optimized lexical analysis component (Blex) provides the necessary balance between applicability and performance;

- *Semantic versus syntactic navigation and binding* Processing a syntax such as IPv4 is semantically driven and requires bindings capable of random access, while processing a streaming syntax such as MPEG video can be driven by the syntax itself, and the bindings can be optimized for the more restricted sequential access;

- *Semantic actions* Semantic variables clarify the grammar (e.g., a semantic variable holding the explicitly computed IP header length), and references in semantic actions can typically be restricted to syntax elements;
- *Symbiosis between inter and intra layer optimizations* Intra–layer attributes are necessary to optimize a composition of layers, while at the same time, feedback from the inter–layer composition can be used to statically optimize the intra–layer bindings.

These general observations should form the starting point for any new language being designed for IO data binding; and would hopefully influence the language used to formally specify an IO data syntax.

7.2 Research Directions

While this dissertation described and addressed several issues regarding the architecture, specification, and implementation of effective IO data access bindings; a variety of interesting questions remain unanswered and offer opportunities for further investigation.

7.2.1 Implementation

The experimental evaluation described in Chapter 5 exposed inefficiencies in the compiled access bindings. For the more complex access structures, the C++ compiler optimizer was unable to resolve pointer aliasing concerns and hence could not traverse branches to the extent necessary to recognize and eliminate duplicate code. In particular the optimizer was unable to distill Xyn's layered *Choice* and *Sequence* implementations into the condition evaluation and pointer manipulations of the low–level C code Xyn is intended to replace. Section 5.3.4 described some avenues to explore that would facilitate the compiler producing more efficient syntax accessors.

The optimization issues faced by Xyn are relevant in a broader scope since the encapsulation (layering) techniques utilized by Xyn are among the abstraction advantages of C++ compared to C. At the same time, C++'s abstractions are intended to exact a minimal performance penalty when compared to functionally equivalent C, hence either Xyn's generated access binding code needs to more closely conform to optimizable abstractions, or maturing C++ compiler technology will accommodate Xyn. The most recent release of the open source Gnu compiler (gcc) successfully compiles Xyn's generated binding code; it is a promising candidate compiler with which to explore further performance enhancements.

Xync

The current Xync (the Xyn compiler) implementation utilizes a parser produced by ANTLR [Parr and Quong, 1995] to build an abstract syntax tree (AST) and then passes this AST to an ANTLR generated tree–walker that instantiates a corresponding Java implemented AST representation. Semantic validation, linking the target syntax declarations and definitions, and binding code generation are performed by the latter (i.e., hand coded Java) AST implementation. The version of ANTLR currently under development supports *heterogeneous* AST nodes — which could accommodate the various Xync AST node types — and code generation facilities. Reimplementing Xync's current back-end to a more capable AST processing framework should result in a more maintainable and extensible tool. One interesting avenue of exploration could be the application of ANTLR's own analysis mechanisms for syntactic ambiguities to Xync's analysis of syntactic predicates to discover ambiguities or missing cases, and duplicate or missing cases in enumerations.

MetaXyn

Chapter 6 described the design of MetaXyn, Xyn's complementary inter–layer optimizer. MetaXyn's deployment depends on the collection of layers comprising the communications stack of an application, from presentation processing to the system network interface, being augmented with Xyn–based intra–layer bindings. MetaXyn also depends on two key IO system technologies that have been the subject of extensive research, but are not yet widely deployed: a user–system boundary crossing implementation supporting copy avoidance, similar to that described in [Brustoloni and Steenkiste, 1996] and an early demultiplexing mechanism such as the one described in [Druschel and Banga, 1996]. Given the above, the meta–layer protocol described in Chapter 6 are relatively straightforward to implement; however, since MetaXyn requires cooperation from each layer along the "vertical cut" through the IO system, its deployment and acceptance in a mainstream system would be considerably more difficult than that of Xyn, which can be independently deployed in any layer.

7.2.2 Application Domains

While the experimental evaluation of Chapter 5 focused on the random access syntax navigation required by the *Internet Protocol* suite, Xyn's origins are actually rooted in the sequential access navigation syntax of multi–media codecs used in presentation layer processing; specifically, the MPEG–1 syntax specification in [ISO/IEC 11172-2]. Prototype C++ binding code for much of the syntax was written as a proof–of–concept; the subsequent evaluation of IPv4 was chosen for the dissertation in order to validate Xyn's random access syntax navigation, in particular, the extreme efficiency demanded by a comparison to C style processing of that protocol. Furthermore, by design, Xyn's random access bindings can largely be directly substituted for the original C bindings in the IO data processing layers, facilitating an accurate performance comparison.

In contrast to the *semantic* or *user directed* transformation between IO data and host types facilitated by random access bindings, a sequential access syntax, such as MPEG, motivates restructuring the codec's operations to put the syntax at the center — an organization this dissertation calls *syntax directed transformation.* This organization follows from both the fixed access pattern inherent in sequentially encoded syntaxes and the hierarchical or layered structure of streaming content encodings. The hierarchical structure of the MPEG video syntax was described in Section 1.1 of the Introduction, and suggests further opportunities to automatically generate processing code from the syntax specification. By organizing the codec's (de)multiplexing around a state machine, i.e., allowing the encoding or decoding to drive the transitions, the structure of the encoding can be validated and the position in the hierarchy tracked. A new operation mode can be implemented by adding transitions that can be selectively enabled by the controlling software. For example, fast forwarding of an MPEG sequence would be implemented by reorganizing the state machine to decode only the first I–picture (i.e., fully intra–coded picture) from each MPEG Group of Pictures (GoP), and then skip IO data elements until the next GoP header. Such organizations could be clearly specified with an extension to Xyn, and the processing code automatically generated.

The potential for enhancing sequentially coded sequences is not limited to the high level organization and processing. In contrast to the myriad calls to “getbits” and “setbits” throughout the typical MPEG codec’s layers, and the implicit transformation between the syntax encoding and domain specific type, bindings generated from a syntax specified using Xyn’s structures and Blex’s extensive catalog of types allows exchanging IO data values with the layer codec in explicit, host defined types. For example, *differential pulse coded modulation* (DPCM) is a technique that represents a current value as a difference from a previous value. Since the difference is expected to be small on average, the field is encoded using an entropy code (a variable length code that utilizes its shortest codes to represent frequently occurring values). Since DPCM codes are used extensively in video codecs, a Blex defined DPCM type is available and is called for in the Xyn specification of the syntax. Then the codec’s interface to the syntax field accessors is in terms of actual difference value, such as a host integer, and the complex entropy coding is hidden within the accessor. DPCM is just one example of a syntax encoding element that could populate an extensive library of reusable elements.

In addition to the above reasons, streaming multimedia codecs are an especially interesting candidate application for Xyn since they are a currently active area of development. New codecs are developed and existing codecs refined to exploit new understandings of human perception of visual and audio data, and to service the wide variety of consumer devices and applications emerging with the widespread deployment of broadband and wireless networking. As described in this dissertation, Xyn could relieve the codec developer of much of the tedium of (re)implementing IO data syntax processing at the granularities of the stream, message, and field.

7.2.3 Language Extensions

Along with specializing Xyn to support application domains utilizing sequential access syntaxes, additional extensions would enhance Xyn’s usability and robustness across domains. In particular, handling of exceptional conditions is left entirely to syntax processing code. For example, when IPv4 encounters a packet with less than 20 bytes (octets) of data behind the network header, it must discard the packet as a "runt" without examining it further, and typically updates a counter. Since a variety of assertions could be explicitly specified in, or even deduced from, IPv4's Xyn syntax specification, a better approach would be to label the exceptional conditions in the specification and allow the IO data processing layer to supply a handler for each condition. In this manner, syntactic constraints are lifted from an implicit representation in the layer implementation to an explicit representation in the specification.

The Xyn language of syntax structure is complemented by the *Binary Lexicon* (Blex), which encapsulates the details of access to IO data's packed, externally typed fields. With the exception of enumeration types, each Blex type is currently implemented as a C++ class, parameterized with the details of its particular instantiation, such as size in bits. The Xyn language compiler (Xync) recognizes a field declared as some Blex type by its distinctive syntax and position in the Xyn specification's parse tree. A corresponding (Java) class in Xync's back-end access–binding code generator emits the field's definition as the appropriately instantiated template class of the requested Blex type. One enhancement proposed for the current implementation is automated range checking, integrated with the error handling described above.

In contrast to the implementation of Blex's elemental types, Xyn generates the binding access code for Blex's enumeration types, such as the IPv4 Type–of–service (ToS) or Option types. Generating additional complex field types could further the utility of Xyn. In particular, optimized accessors for types such as DPCM (discussed above) and other entropy encodings could be generated from the specification: typically a per–value mapping between the bit representation and corresponding host type; sometimes including an "escape" code to represent statistically infrequent values.

Another candidate enhancement is a Blex language processor for composing complex Blex types from other Blex primitive types; such as translating between a host floating point type and a sign–exponent–mantissa representation encompassing three primitive fields in the syntax. The latter is currently done directly within the C++ implementation; however, operating at a more abstract level would simplify adding new types to the Blex library.

Appendix A

ip.xyn: A Xyn Specification of IPv4

This appendix presents the *Internet Protocol, version 4* (IPv4) [Postel, 1981a], the internetworking layer of the TCP/IP suite, specified using the Xyn language. The entire specification including all options would be excessively long, so a representative sample is shown here. This sample encompasses the examples used in the dissertation and includes the timestamp option, which is one of the more complex options.

```
syntax IPv4Net ;

xynOptions {
    access_style = "random" ;
}

IP
  :  // 1. Protocol layer processing view (header only).
     ip_v   : 0b0100          // IP version 4
     ip_hl  : UIMSBF<4>       // [,5..15]
     ip_hl_bytes : { xyn_scope.ip_hl * 4 } // header length in bytes
     size   : { xyn_scope.ip_hl_bytes * 8 } // size of header bits (required)
     ip_tos : BSLBF<8>        // type of service XXX: more detail
     ip_len : UIMSBF<16>      // [,20..65535] total length in Octets
     ip_id  : BSLBF<16>       // identification
     ip_RF  : 0b0             // reserved (flag)
     ip_DF  : Boolean         // don't fragment
     ip_MF  : Boolean         // more fragments
     ip_off : UIMSBF<13>      // fragment offset
     ip_ttl : UIMSBF<8>       // time-to-live
     ip_p   : BSLBF<8>        // protocol
     ip_sum : UIMSBF<16>      // checksum
     ip_src : Address         // source address
     ip_dst : Address         // destination address
     ip_options_bytes : { xyn_scope.ip_hl_bytes
                          - xyn_scope.ip_options_offset / 8 }
     ip_options : ( { xyn_scope.ip_options.SizeBits() / 8
                      < xyn_scope.ip_options_bytes }
                       => opt:Option[ xyn_scope.ip_options_bytes
                                      - xyn_scope.ip_options.SizeBits() / 8 ] )*
  ;

//
//  Local elements.
//

protected
Octet
  :  BSLBF< 8 >
  ;

protected
Address
  :  // Note: class A, B, C, ... could be distinguished syntactically
     // using literal (prefixes), but CIDR likley makes that pointless.
     a : Octet
     b : Octet
     c : Octet
     d : Octet
  |  // N.B., host order.
     s_addr : BSLBF<32>  // XXX: fast equality tests;
  ;

//
// IP Options
//

protected
Option[ int bytesLeft ]
  : // Give an Octet worth of literal for efficiency
    // nop is used for internal padding to 4-octet boundary
    ( OCopy.no OClass.control OType.nop )
       => nop:( a:OCopy.no b:OClass.control c:OType.nop)
//| ... other options ...
  |
    ( OCopy.no OClass.debmeas OType.ts )=> ts:TS
//| ... other options ...
  | // If nothing else matches and space left, then 0x00s to pad to
     end:( a:OCopy.no b:OClass.control c:OType.end )[ bytesLeft ]
  ;

// Shared option prefix elements.
```

```
protected
OCopy<1,enum>
  :  no:0b0
  | yes:0b1
  ;

protected
OClass<2,enum>
  : control:0b00
  | debmeas:0b10
  ;

protected
OType<5,enum>
  :  end:0b00000  // end-of-options list
  |  nop:0b00001  // intra-option pad
  |  sec:0b00010
  | lsrr:0b00011  // loose source routing, record route
  |   ts:0b00100  // timestamp
  | esec:0b00101
  | csec:0b00110
  |   rr:0b00111  // record route
  |   si:0b01000
  | ssrr:0b01001  // strict source routing, record route
  |   xm:0b01010
  | mtup:0b01011
  | mtur:0b01100
  |  xfc:0b01101
  |  xac:0b01110
      // 0b01111
  |  imi:0b10000
  |  eip:0b10001
  |   tr:0b10010
  |   ae:0b10011
  |   ra:0b10100
  | sdbm:0b10101
  | nsap:0b10110
  |  dps:0b10111
  |  ump:0b11000
  ;

// Options with extended parts.
protected
TS
  : a:OCopy.no
    b:OClass.debmeas
    c:OType.ts
    length:UIMSBF<8>     // [4..40] entire option in Octets
    size:{ xyn_scope.length * 8 }
    pointer:UIMSBF<8>    // [5..(length+1)] Octet offset to next available
    overflow:UIMSBF<4>   // incremented at each node where pointer > length
    flags_u0:0b0          // reserved
    flags_u1:0b0          // reserved
    flags_isPrsp:Boolean  // timestamp contains prespecified address fields.
    flags_isAddr:Boolean  // timestamp contains IP addresses, also.
    list_sizeOctets:{ ( xyn_scope.size
                        - ( xyn_scope.list_offset
                            - xyn_scope.a_offset ) ) / 8 }
    list_elementOctets:{ xyn_scope.flags_isAddr ? 8 : 4 }
    list_elementCount:{ xyn_scope.list_sizeOctets
                        / xyn_scope.list_elementOctets }
    list_elementIndex:{ ( xyn_scope.pointer - 4 - 1 )
                        / xyn_scope.list_elementOctets }
    list:(
       {xyn_scope.flags_isAddr}=> avec:(
                 node : Address
                 isNS : Boolean         // !(time IS ms since midnight UT)
                 time : UIMSBF<31>      // msecs since midnight UT or arbitrary
           )[ xyn_scope.list_elementCount ]
      |
        sans:(
                 isNS : Boolean         // !(time IS ms since midnight UT)
                 time : UIMSBF<31>      // msecs since midnight UT or arbitrary
           )[ xyn_scope.list_elementCount ]
     )
  ;
```

Bibliography

Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992. URL `citeseer.nj.nec.com/batory92design.html`.

J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *The Second Symposium on Operating Design and Implementation*, pages 277–291. USENIX Association, Berkeley, CA, USA, 1996.

CCITT. Specification of Abstract Syntax Notation One. International Telegraph and Telephone Consultative Committee, 1988. Recommendation X.208.

D.D. Clark and D.L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols, Philadelphia, PA, Sept. 1990*, pages 200–208. ACM, 1990.

S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification, 1998. URL `http://www.ietf.org/rfc/rfc2460.txt`. RFC 2460. Available from `http://www.ietf.org/rfc/rfc2460.txt`.

Tim Dierks and Christopher Allen. The TLS protocol version 1.0. Internet Draft, November 1998. URL `http://www.ietf.org/internet-drafts/draft-ietf-tls-protocol-06.txt`. Expires May 12, 1999.

E. W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the Association of Computing Machinery*, 11(5):341–346, May 1968a.

E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976b.

P. Druschel and L. Peterson. Fbufs: A high–bandwidth cross–domain transfer facility. In *Proceedings 14th Symposium on Operating System Principles*, pages 189–202. Association for Computing Machinery, December 1993.

P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings, 1994 SIGCOMM Conference*, pages 2–13, London, UK, August 31st - September 2nd 1994.

Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 261–275, Berkeley, CA, USA, October 1996. USENIX.

Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 44–56, New York, June 15–18 1997. ACM Press.

D. R. Engler, M.F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application–level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM, December 1995.

Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In USENIX Association, editor, *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 327–333, Berkeley, CA, USA, Winter 1993. USENIX. ISBN 1-880446-48-0.

Stephen P. Hufnagel and James C. Browne. Performance properties of vertically partitioned object-oriented systems. *IEEE Transactions on Software Engineering*, 15(8):935–946, August 1989.

ISO. Information Processing — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). International Organization for Standardization and International Electrotechnical Committee, 1987. International Standard 8825.

ISO7498. ISO7498: Basic Reference Model for Open Systems Interconnections (CCITT Recommendation X.200). Technical report, ISO/CCITT, 1988.

ISO/IEC 11172-2. Information technology – coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s – video, 1993.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer. ISBN ISBN 3-540-63089-9.

Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000. ISSN 0734-2071. URL `http://www.acm.org/pubs/citations/journals/tocs/2000-18-3/p263-kohler/`.

R. Greg Lavender, Dennis G. Kafura, and R. W. Mullins. Programming with ASN.1 using polymorphic types and type specialization. In Manuel Medina and Nathaniel S. Borenstein, editors, *ULPAA*, volume C-25 of *IFIP Transactions*, pages 151–166. Elsevier, 1994. ISBN 0-444-82047-7.

Microsoft, Inc. Windows NT TransmitFile system call, 1996. URL `http://www.microsoft.com/msdn`. Win32 (NT specific) interface documentation.

Object Management Group. Common Object Request Broker Architecture: Core Specification, December 2002. URL `http://www.omg.org/cgi-bin/doc?formal/02-12-06`.

Sean O'Malley, Todd Proebsting, and Allen Brady Montz. USC: A universal stub compiler. In *Proc. Conf. on Communications Archi. Protocols and Applications*, London (UK), September 1994.

J.K. Osterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000. Available online at `http://www.acm.org/pubs/citations/journals/tocs/2000-18-1/p37-pai/`.

Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software — Practice and Experience*, 25(7):789–810, July 1995. ISSN 0038-0644.

Joseph Pasquale, Eric Anderson, and P. Keith Muller. Container Shipping: Operating System Support for I/O-Intensive Applications. *Computer*, 27(3):84–93, March 1994.

David C. Plummer. An Ethernet Address Resolution Protocol. Request for Comments 825, DDN Network Information Center, SRI International, September 1982.

J. Postel. User datagram protocol. RFC 768, Internet Society (IETF), 1980. URL `http://www.ietf.org/rfc/rfc768.txt`.

J. Postel. RFC 791: Internet Protocol, September 1981a. URL `ftp://ftp.isi.edu/in-notes/rfc760.txt,ftp://ftp.isi.edu/in-notes/rfc791.txt`. Obsoletes RFC0760. Status: STANDARD.

J. Postel. Transmission control protocol. RFC 793, Internet Society (IETF), 1981b. URL `http://www.ietf.org/rfc/rfc793.txt`.

Schulzrinne, Casner, Frederick, and Jacobson. RTP: A transport protocol for real-time applications. *Internet-Draft ietf-avt-rtp-new-01.txt (work inprogress)*, August 1998. URL `ftp://ftp.ietf.org/internet-drafts/draft-ietf-avt-rtp-new-01.txt`.

R. Srinivasan. RFC 1833: Binding protocols for ONC RPC version 2, August 1995. URL `ftp://ftp.internic.net/rfc/rfc1833.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1833.txt`. Status: PROPOSED STANDARD.

Sun Microsystems, Inc. RFC 1014: XDR: External Data Representation standard. Internet Draft, IETF, June 1987. URL `ftp://ftp.internic.net/rfc/rfc1014.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc1014.txt`. Status: UNKNOWN.

Sun Microsystems, Inc. Java RMI, 1996. URL `http://java.sun.com/products/jdk/rmi/`.

Sun Microsystems, Inc. Java object serialization specification. Copyright 1996–2001, Sun Microsystems, Inc., August 2001. URL `http://java.sun.com/j2se/1.4.1/docs/guide/serialization/index.html`.

Gnanasekaran Swaminathan. CUG400 — Socket++. *C Users Journal*, 12(4):121–128, April 1994. ISSN 0898-9788.

Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: a user-level network interface for parallel and distributed computing. *ACM Operating Systems Review, SIGOPS*, 29(5):303–316, 1995.

Erik Walthinsen. gstreamer Open Source Multimedia Framework. http://www.gstreamer.net/, 1999.

Dave Winer. Four years of xml–rpc, 2002. URL `http://davenet.userland.com/2002/04/04/fourYearsOfXmlrpc`.

F. Yergeau. RFC 3629: UTF-8, a transformation format of ISO 10646, November 2003. URL `ftp://ftp.internic.net/rfc/rfc3629.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc3629.txt`. Obsoletes RFC2279. See also STD0063. Status: STANDARD.

Vita

David Scott Page was born in 1959. He is the son of Howard Page and Carol Page. He joined the University of Central Florida in 1987 and received the Bachelor of Science in Computer Science 1989, and Master of Science in Computer Science in 1990. He joined the University of Texas at Austin in 1992 and received the Master of Science in Computer Sciences from there in 1998.

Permanent Address: 1507 West 30th Street
Austin, TX 78703

This dissertation was typeset with $\LaTeX\,2_\varepsilon$[1] by the author.

[1] $\LaTeX\,2_\varepsilon$ is an extension of $\LaTeX$. $\LaTeX$ is a collection of macros for $\TeX$. $\TeX$ is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.