RC23103 (W0402-026) February 5, 2004
Computer Science

IBM Research Report

Achieving Scalability and Throughput in a Publish/Subscribe System

Mark Astley, Joshua Auerbach, Sumeer Bhola, Gerard Buttner, Marc Kaplan, Kevan Miller, Robert Saccone, Jr., Robert Strom, Daniel C. Sturman, Michael J. Ward, Yuanyuan Zhao
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Achieving Scalability and Throughput in a Publish/Subscribe System

Mark Astley, Joshua Auerbach, Sumeer Bhola, Gerard Buttner, Marc Kaplan, Kevan Miller, Robert Saccone Jr., Robert Strom, Daniel C. Sturman, Michael J. Ward, Yuanyuan Zhao

Abstract— **We describe the Gryphon content-based publish/subscribe messaging system. The objective of Gryphon is to provide the capabilities of message oriented middleware systems in widely distributed and high-volume environments, i.e. distributed across countries or continents, with tens of thousands of messaging clients and with tens of thousands of messages being delivered across the system each second. We introduce the essential design concepts of Gryphon, with particular emphasis on the decisions that led to high throughput and scalability. We present performance benchmarks that demonstrate the effectiveness of these decisions.**

Keywords— **Publish/Subscribe, Content-Based Routing, Scalable Overlay Network**

I. Introduction

PUBLISH/SUBSCRIBE messaging (pub/sub) has emerged as a popular paradigm for building asynchronous distributed applications [1], [2], [3], [4], [5], [6], [7]. A pub/sub system consists of publishers that generate messages and subscribers that register interest in all future messages matching the conditions specified in their subscription. The system is responsible for routing published messages to interested subscribers. Information providers and consumers are decoupled, since publishers need not be aware of which subscribers receive their messages, and subscribers need not be aware of the sources of the messages they receive.

Gryphon is an implementation of pub/sub on an overlay network of *broker* machines. It is designed for high-volume, low latency, Internet-scale distribution, that is, a target deployment of multiple brokers distributed across countries or continents, with tens of thousands of messaging clients and with tens of thousands of messages being delivered across the system each second. Gryphon is a *content-based* publish/subscribe system, which means that subscriptions to messages are not limited to a set of predefined topics but may additionally specify predicates on message contents.

In this paper, we present the design of the Gryphon publish/subscribe messaging system, emphasizing those features that enable Gryphon to support high throughput in its target environments.

Next, we focus on two aspects of system behavior that are critical to Gryphon scalability and performance.

- I/O SCHEDULING: We discuss how we manage concurrency to exploit processor power on SMP architectures, while avoiding reduced performance due to excess context-switch time. The I/O system has also been optimized to distinguish between broker-to-broker links (usually a small number of fully utilized links) and broker-to-client links (usually a large number of intermittently utilized links).

The authors are employed by the IBM T. J. Watson Research Center, Yorktown Heights, New York, USA.

- ACCESSING MESSAGE CONTENT: Messages arriving at a Gryphon broker are filtered by being matched against content predicates and security policies, and are then forwarded to appropriate outbound links. To match a message in a content-based system, the relevant fields must first be extracted, and then the content filters must be evaluated. In an earlier paper [8], we discuss how we use an optimized content filter evaluation data structure to minimize the time to match a single message against a large number of subscriber-defined content predicates. In this paper, we describe how we encode structured messages defined in a flexible schema definition language in a way that allows fast lookup of individual fields during the content matching step.

Finally, we present an experimental analysis that measures the performance of Gryphon and validates our design decisions.

II. Gryphon Approach and Concepts

In this section, we introduce the concepts and architecture of Gryphon.

A. Client and Broker Roles

Gryphon is a *broker-based* messaging infrastructure over an *overlay network*. That is, within the Gryphon system, there are two primary roles: a *client* that produces and/or consumes messages, and a *broker* that provides the majority of messaging function and is responsible for moving data through the network. This approach is in contrast to a message bus or peer-to-peer approach where almost all messaging function is included at each client node, and clients connect directly to each other. Gryphon clients are light-weight, with the majority of function being provided at the broker.

Both client-to-broker and broker-to-broker logical links are implemented over TCP/IP connections. In all cases, there is only a single, bi-directional connection between any broker and any other broker or client. An alternative approach would have been to build a custom point-to-point protocol on top of UDP. Since we needed many of the services of TCP, such as fragmentation and windowing, we chose to use standard TCP. Use of TCP significantly reduced our development cost, and additionally allowed us to work smoothly with firewalls and other security standards that expect TCP. As we show in Section V, we were still able to achieve very high performance building on TCP.

B. Broker topology

There are two topology concepts in the description of Gryphon's overlay broker network: *cells* provide a grouping mechanism for brokers and *link bundles* provide a group mechanism for connections between cells.

A cell represents a cluster of nearby brokers that can share load and can provide coverage in case of failure. We require that the set of live brokers within a single cell be fully connected. A physical broker may participate in more than one cell; in that case, the physical broker is viewed as multiple *virtual brokers*. An optimization exists to avoid having to send messages over a physical link when two virtual brokers share the same machine. Another optimization exists so that messages destined to multiple virtual brokers in an adjacent physical machine can travel in a single message.

A *link bundle* is the collection of links between a pair of cells. The link bundle is a logical representation of all the links between brokers that may be used to route messages from one cell to another. Link bundles provide load balancing and redundancy.

The Gryphon topology is built using cells and link bundles. At any given time, a single Gryphon spanning tree is used to route each message. This spanning tree is constructed over the graph of cells and link bundles, where cells are nodes in the graph, and link bundles the arc in the graph. The brokers then use the spanning trees to route messages through the network, while still having the flexibility to choose from a redundant set of routes for failure response and load balancing purposes. Therefore, failures and reconfigurations are handled locally: when a broker fails, traffic is re-routed through another broker in that cell, and when a link fails, traffic is routed through another link in that link bundle (either directly from the current broker or indirectly via another broker in the same cell.) A key design principle behind Gryphon is that control and recovery traffic should almost always be localized.

There is a mechanism by which administrative entities can dynamically change the topology: namely, to change the membership of brokers within cells, to change the neighborhood relationships among cells, and to change the spanning trees in response to such changes. There are also mechanisms for having multiple spanning trees, although any message entering the system is routed on exactly one of them.

In figure 1 we show an example of a Gryphon topology.

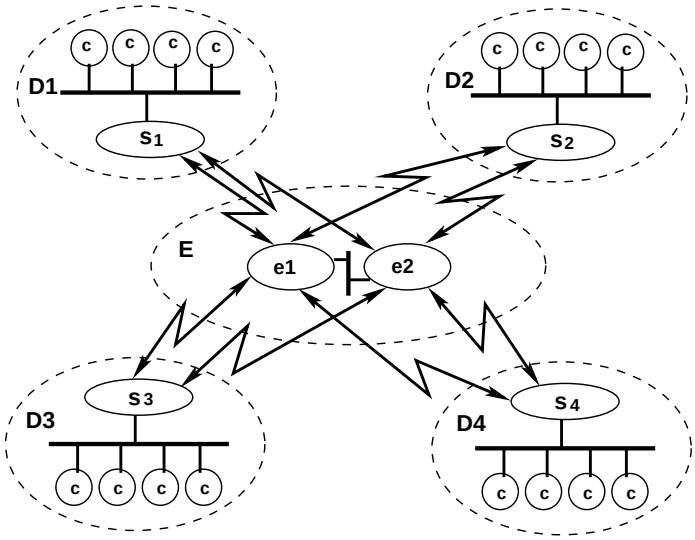

Fig. 1. An example Gryphon topology. In this example, there are five cells: D1 through D4, and E. Each of the "D" cells consists of one or more "small" servers (the "c" machines) and a "larger" routing server (the "s" machines) which routes local messages to the rest of the network. Cell E is a "backbone" cell which contains two large routing servers e1 and e2. The link bundles are defined between the D's and E such that one of the backbone servers may fail without disrupting routing. A single spanning tree (not shown) is defined for this network and is identical to the cell/link bundle topology.

C. Client Interface

Client applications first establish a TCP/IP *connection* to a Gryphon broker; once connected (and authorized by Gryphon's security functions), they may then publish messages or register subscriptions to receive future messages.

Clients may define message formats using a specification called a *schema*. Library functions allow publishers to write and subscribers to read fields of messages by name. Schemas are administratively associated with *topics*. Content-based subscriptions to a particular topic can specify content selection predicates using the names defined in the topic's schema. As we shall discuss later, Gryphon uses the schema specification to determine a byte encoding for messages that is optimized for efficient extraction of values as messages are routed and filtered through the broker network.

III. Efficient I/O Scheduling

The most frequent task performed by a broker is to read a message from an I/O connection, match the message, and send it out on all connections with subscriptions that match the message. A connection may be to an adjacent broker or to a client connected to this broker. The following characteristics of a multi-broker deployment motivate the I/O features implemented in the broker:

1. The number of clients connected to a broker can be very large, bounded only by the operating system's limit on number of sockets. These clients can have high variability in capacity and rate of reading and writing — for instance, some clients may be only publishing, others only subscribing.
2. Since messages may be matched against complex content filters, the processing overhead of each incoming message is non trivial. Also, in many deployments, most data messages may be flowing in a certain direction in the topology tree, so certain broker connections may have more incoming messages than others. These reasons motivate decoupling reading a message from its processing, and utilizing parallelism in processing.

The above concerns motivate careful management of threads to exploit the intrinsic parallelism in machines, and to reduce the overhead associated with thread switching.

Gryphon incorporates a thread scheduling component containing a thread pool, whose threads are assigned to do reads and writes on client connections. The read and write operations are non-blocking, so the thread may be returned to the pool after partially completing a read/write. The underlying TCP sockets are monitored by a scheduler thread to detect whether they are *ready-to-read* or *ready-to-write*. If a socket is ready-to-read, a thread is assigned (from the thread pool) to do a non-blocking read. If a socket is ready-to-write, and there is data waiting to be sent out on this socket, a thread is assigned to do a non-blocking write. The thread that finishes reading a message from a socket then matches the message against the content filters, and for

the matching output connections, enqueues it on outgoing per-connection output queues.

For server-server connections, the thread pool is used to perform socket writes, while a dedicated thread performs socket reads. Socket reads are optimized for throughput as described in Section III-C below.

The socket monitoring function is implemented using native code. In Unix variants, it is implemented using the `poll()` system call. The Windows I/O Subsystem is asynchronous by design and it is because of this that a combination of non-blocking and asynchronous I/O is used to maximize scalability. Windows asynchronous I/O is a mechanism which allows the thread that issues the I/O to continue execution while the system performs the I/O operation. Notification of completion of the I/O operation occurs via a callback function specified when the operation was initiated. Completion notifications are dispatched to threads that are part of a thread pool created and maintained by the operating system on behalf of the process.

A. Cut-thru Write Optimization

When a new message needs to be written to a socket, the common case is that all previous messages have been written by the thread pool to the socket (the output queue is empty), and the OS has enough buffer space available to accept the new message. In this common case, the non-optimized processing path described earlier is inefficient, since it results in extra thread context switches to do the write, and also increases end-to-end message latency in the system.

With *cut-thru writes*, the thread doing the matching does a non-blocking write on all matching connections whose output queue is empty and whose socket is ready to write data. If the write completes, nothing needs to be queued on the output queue. However, if the message is only partially written because the socket cannot accept the full message without blocking, the remainder of the message is queued on the output queue, and will be eventually written by a thread from the thread pool. On Windows, any data that cannot written as part of the non-blocking write will result in an asynchronous write operation being initiated for the remaining data.

B. Socket Hierarchies for Efficient Input Monitoring

The `poll()` system call requires the list of sockets which are being monitored, represented as a list of Unix file descriptors, to be passed in as a parameter to each call. This has two problems: (1) the large size of the parameter list — possibly tens of thousands of file descriptors, and (2) the fact that some sockets are frequently ready to read (and should be frequently polled) and others not.

To optimize for this case, the list of sockets is automatically organized into a 2-level hierarchy, where descriptors in the top level are frequently ready to read. File descriptors are moved from the top level to the bottom level if the number of messages coming in on that connection decreases and vice versa. The bottom level of this hierarchy contains more than one list since we want to break up the typically large list of infrequent publishers into smaller lists.

Once the list is broken up into multiple lists, the broker assigns a thread to monitor each list independently. Note that since the `poll()` call is used, monitoring of a list in the lower level is not less frequent than the list at the top level. However, there is a significant performance improvement since the `poll()` call will return much less frequently for such a list.

On Windows, the poll list and socket hierarchies are not used to determine when data is available to read. Instead an asynchronous read is issued for each socket. The asynchronous read will only complete when data has been read from the socket and it will be delivered to the callback notification function running on one of the worker threads in the system managed thread pool.

C. Input Queuing on Broker Connections

Each broker connection is associated with a dedicated read thread for reading messages from the connection. In the absence of input queuing, this thread performs the matching step for the message and either queues the message on out-buffers or does cut-thru writes. If the input rate is sufficiently high, this dedicated thread is not able to keep up with the input rate. Input queuing solves this problem by utilizing parallelism in the broker while ensuring that messages typically maintain publisher order as they travel through the system. [1]

Input queuing is configured using a count, n, representing the number of input queues per connection, say numbered $0...n-1$. The dedicated read thread reads messages from the connection, and hashes the publisher identifier (an integer) to yield a number in the range $0...n-1$. (A simple randomized universal hash function requiring only one multiply and several add and shift operations is used.) The message is enqueued on the queue represented by this number. Each of the n input queues has a dedicated thread that is responsible for dequeuing the next message from the input queue, and performing the matching and output steps.

IV. Gryphon Message Format

A. Motivation and Context

As described in [8], Gryphon achieves sublinear time scaling and linear space scaling as the number of subscriptions increases. This approach, however, requires that values be extracted from arbitrary message fields in constant time. Since messages are complex nested information structures, extracting values in constant time is a challenge. In addition, we want the extraction time to be as low as possible, given that messages must start out and end up as byte arrays, that only a small fraction of each message is typically accessed to make routing decisions, and that an even smaller amount of each message will be modified before retransmission. These considerations gave rise to the design of the Gryphon message format.

We quickly rejected any format that requires the message to be deserialized into a non-linear graph of objects and reserialized if modified (e.g., the standard Java serialization format). Most such processing would be wasted on typical messages. We considered two approaches: tagged formats, such as XML, and untagged binary formats, such as those used in IP and TCP networking layers.

[1] Enforcing publisher order is still the responsibility of the receiving endpoint, since failures could cause messages to get out-of-order.

A.1 Tagged Formats (XML)

Tagged formats have the advantage that they are readily extended as the system evolves. XML is also quite good at expressing nesting, and so can capture the contribution of different protocol and software layers to the composition of message. It is possible to examine or even modify an XML message while leaving it in serial form. But, to find arbitrary information that is deeply buried in the tag structure of an XML document requires a linear scan over the document and some relatively expensive string-based parsing. XML is also an extremely bulky format, which is problematic given that, in a pub/sub system, message size is inversely related to system throughput.

A.2 Untagged Binary Formats

Constant time access to a flat series of fields is readily achieved by designing tailored binary message layouts (which are also compact). Such formats are commonly used by lower networking protocol layers (e.g., IP and TCP). Fields of varying length can be placed after fields of fixed length and offsets to fields whose offset would not otherwise be predictable can be stored at predictable offsets. With this type of design it is possible to ensure that every field's offset is known after reading at most one indirect offset from the message.

Untagged binary formats have three problems.

1. Creating them, and the code to use them, is labor intensive.
2. Access time increases with nesting of information, which is inevitable in a complex system. Access to header fields may take constant time, but it requires stepping into another format definition in order to process the payload (which may itself be a nested structure, so it may take several such steps to reach the important information).
3. Untagged formats do not provide a convenient evolution mechanism. A process expecting version N of the format will be baffled when confronted by version $N + 1$. A process that wants to support both versions will typically require case logic.

A.3 The Gryphon Binary Format

The advantages of a binary format are so substantial that we adopted that approach, taking specific steps to overcome the three deficiencies mentioned. The first problem is addressed through the well-known technique of defining the messages in a schema language and using a tool to translate the schema into runtime artifacts. Our key contributions lay in solving the other two problems.

1. We transform the "natural" schema (which makes the nesting of information explicit for human understanding) into an isomorphic derived internal schema that flattens the nested information structures. Constant-time access is usually achievable with this transformation (limitations are discussed below). Our tools and runtime hide the fact that the schema has been transformed.
2. We use a novel approach to schema evolution that provides a degree of extensibility equivalent to what can be achieved with tagged formats, even though we use a compact untagged format that is tuned for efficiency.

This paper will emphasize the first contribution. We also provide some information about the second in order to clarify that one of the key advantages of XML (extensibility) was not sacrificed in the process.

B. Gryphon Schemas

Gryphon uses its own schema language internally, rather than adopting a standardized one, so that the schema transformations that it needs can be easily carried out. However, Gryphon schemas can be the translation targets of other schema declaration mechanisms such as UML, XML Schema, or ASN.1.

A Gryphon schema is a tree made up of atomic, tuple, list, and variant types, schema references, and the special type **dynamic**.

B.1 Atomic Types

The precise repertoire of atomic types was chosen pragmatically and does not matter to other aspects of the solution. In this paper we use a few representative types such as **int**, **string**, and **boolean**.

B.2 Tuple Types

A tuple type is an indexed set of fields where the fields can be of any type (atomic or aggregate). Tuple types are identified by listing their field types between square brackets.

```
[ int , [ string , boolean ] ]
```

The empty tuple type **[]** is permitted.

B.3 List Types

A list type denotes zero or more repetitions of its item type, which can be any non-vacuous type. Lists are identified by enclosing the item type in special parentheses, e.g. ***(int)***, ***([int,boolean])*** etc.

B.4 Variant Types

A variant type is a choice amongst alternative cases, which can be of any type. Variant types are identified by listing their cases between curly braces separated by bars.

```
{ [] | int | *(string)*}
```

A variant type must have at least one case.

Variants are important in message schemas because of their role in managing layered sets of protocols. What is a "payload" or "body" at one protocol level has an enumerable set of alternative definitions at the next higher level, depending on the type of payload in question. Gryphon allows variants to be extended with more alternatives, as part of its evolution mechanism, so this is not a deterrent to using them to enumerate all the cases that are presently known.

B.5 Dynamic Types

A dynamic type (identified by the keyword **dynamic**) represents a point in a schema where something conforming to a different schema (not yet known) will be substituted at runtime. Access to information in a dynamically typed field is not constant (it takes time to step into a dynamic field, just as it takes time to step from the IP packet into the enclosed TCP packet when using traditional binary formats).

B.6 Schema Names and Schema References

Once a schema is given a name, fields can then be declared has having that schema, thus incorporating one schema within another. Recursive references are possible, but the recursive incorporation of a schema uses the same runtime mechanism as **dynamic**, and thus does not achieve constant time reference.

C. The Flat-Tuple Transformation

The purpose of the flat-tuple transformation is to take a schema that is readily understood by humans and derive from it a schema that can achieve constant time access to fields whose declaration in the original schema may be deeply nested and/or conditioned on the case settings of variants.

C.1 The Theoretical Basis

The transformation is based on established work in the isomorphisms of types [9], [10]. In this body of work, a set of axioms establishes equivalence, under isomorphism, between pairs of types. Inverse transformations always exist between values of isomorphic types. Accessors and mutators that expect a particular type can be transformed into equivalent operations on the isomorphic type. Type isomorphisms have been used in the past to support flexible retrieval of components from libraries [11], [12] and automated generation of converters and adapters [13], [14].

In a type system containing tuple types and variant types, the important rules are the following.

1. Tuple and variant types each obey an associative law. For example,

```
[ int, [ string, boolean ]]
 <=> [ int, string, boolean ]
```

and

```
{{ [] | string } | *(string)* }
 <=> { [] | string | *(string)* }
```

2. Tuple types distribute over variant types. For example,

```
[ int, { string | boolean } ]
 <=> { [ int, string ] | [ int, boolean ] }
```

In a type system containing tuples and lists, there is an additional morphism that is available.

3. A list of tuples can be encoded as a tuple of lists. For example,

```
*([ string, int])* => [ *(string)*, *(int)* ]
```

This is an isomorphism only under the constraint of equal length for the lists. However, it works for our purposes because we only apply it in one direction (which results in the constraint always being met).

By repeatedly applying the associative rules in the flattening direction, the distributive rule in the distributing direction, and the list/tuple rule in its only available direction, one can derive a schema whose values encode the values of any actual schema but that takes on the following constrained form:

- Every tuple type is either at top level or is directly enclosed in a variant (tuples enclosed in tuples were eliminated by the associative rule for tuples and tuples enclosed in lists were eliminated by the list/tuple rule).
- Every variant is either at top level or is directly enclosed in a list (variants enclosed in variants were eliminated by the associative rule for variants and variants enclosed in tuples were eliminated by the distributive rule).

C.2 The Optimal Outcome

If there are no variants enclosed in lists (directly or indirectly) in the original schema, and no recursive schema references, the derived schema that results from a flat-tuple transformation will have the following components.

- At most one variant type (at top level). We call the top-level variant of the transformed schema the *super-variant*.
- All tuple types (if any) are directly enclosed in the super-variant. We call the the tuples that are directly enclosed in the super-variant *flat-tuple types*. If there is no super-variant, then there is only one flat-tuple type and it is the top-level type.
- Everything else is either a basic type (atomic or **dynamic**) or a nested stack of list types enclosing a basic type.

For example, consider the following schema:

```
com.ibm.gryphon.envelop: [
  priority: int,
  properties: *(
    [
      name: string,
      value: string
    ]
  )*,
  payload: {
    absent: [] |
    subscribeRequest: [
      topic: string,
      selector: string
    ] |
    subscribeReply: int |
    other: dynamic
  }
]
```

Its flat-tuple transformation (ignoring non-leaf names) is as follows:

```
{
  [
    priority: int,
    id: long,
    *( name: string )*,
    *( value: string )*
  ] |
  [
    priority: int,
    id: long,
    *( name: string )*,
    *( value: string )*,
    topic: string,
    selector: string
  ] |
  [
    priority: int,
    id: long,
    *( name: string )*,
    *( value: string )*,
```

```
    subscribeReply: int
  ] |
  [
    priority: int,
    id: long,
    *( name: string )*,
    *( value: string )*,
    other: dynamic
  ]
}
```

The transformation into super-variant/flat-tuple form means that the message can be represented as follows.

- There is a single integer enumerating which of the super-variant's flat-tuple cases is the active one. We call this the *multi-choice code*.
- This is followed by a straightforward binary offset encoding of the fields that are present in the selected flat-tuple case.

Constant time access is thereby achieved, certainly, for fields that are not lists. The message may still contain (possibly nested) lists, and access to these is not quite constant. The fixed length elements of a list of such are accessed in constant time by multiplying the index position by the element size, but this is only after locating the start of the list by the usual mechanism. Similarly, the variable length elements of a list of such are accessed in constant time by using an offset table at the start of the list. So, the access time increases based (only) on the number of nested lists and whether each list is fixed or varying. The cost of stepping into a list is less than, but comparable to, the cost of stepping into a dynamic field.

The ability of the flat-tuple transformation to completely flatten a schema is spoiled by the presence of either recursion or of variants that are dominated by lists. We consider these hard cases next.

C.3 Recursion

Recursion is handled by replacing the recursive schema reference with a **dynamic**. This pretends that we don't know the schema until runtime. We actually do, we are merely using this device to avoid an infinite regress during flat-tuple transformation. But, having made the substitution, we are forced to use the dynamic mechanism at runtime, and we give up on constant time access.

C.4 Variants Dominated By Lists

Variant types that are dominated by lists will end up getting "stuck" under the list during the transformation because there is no rule of isomorphism that enables further progress. The best we can do is to perform a secondary flat-tuple transformation on everything under the list-variant node pair as if the variant were the root of its own schema. The field that results in the message is called a *variant box* and it gets its own multi-choice code and binary encoded flat-tuple. The cost of stepping into a variant box is roughly the same as the cost of stepping into a dynamic field (bearing in mind that such fields are always in lists, so there is list indexing overhead as well).

In order to minimize the number of fields that get caught in variant boxes, we apply the list/tuple rule of isomorphism to a list before applying the distributive rule to anything enclosed in the list. For example, if we had

```
*( [ string , { int | boolean } ] )*
```

we would transform it to

```
[ *(string)* , *( { int | boolean } )* ]
```

rather than

```
*( { [string , int ] | [ string , boolean ] } )*
```

Distributing tuples over variants too eagerly, as in the third example, results in more fields getting caught in variant boxes. In the example just shown, the string escaped from the box by getting its own list.

C.5 A Practical Implementation

The previous section describes the flat-tuple transformation as a potentially static operation on schemas. It could be implemented that way, but such an implementation is not necessarily practical. It should be clear that a schema gets larger in the process of distributing its variants. If the schema has many independent variants, you can get a combinatorial effect. For example,

```
[ { w:int | x:int },
  { y:int | z:int } ]
```

is transformed to

```
{ [ w:int, y:int ] |
  [ w:int, z:int ] |
  [ x:int, y:int ] |
  [ x:int, z:int ] }
```

In the Gryphon system schema, for example, the super-variant has several thousand cases. Each case has a certain amount of additional metadata (offset tables, variant case setting maps, etc.) that are used to speed up runtime decoding of the actual flat-tuple. So, the memory footprint and load time for the complete transformed schema could be considerable. Every independent variant that is added to the schema multiplies the size of this footprint by the number of cases in the variant.

But, in typical usage patterns, only a modest fraction of those cases are used at all, and some of those very infrequently. So, most of the transformed schema would be wasted space if the transformation were applied eagerly. In fact, each flat-tuple case represents a concrete assemblage of fields in a message that might occur in practice. It turns out that we can wait until that combination of fields actually does occur before doing the requisite part of the transformation.

Gryphon's incremental runtime form of the flat-tuple transformation keeps the original untransformed schema in memory. The transformation is then applied one flat-tuple case at a time as messages conforming to that case are encountered for the first time. The metadata for flat-tuples that have been encountered before are retained to amortize the cost of producing them. If the amount of retained metadata were to become too large, an LRU policy could be used to discard little-used cases (Gryphon does not currently implement LRU discarding).

D. Achieving Extensibility

Gryphon makes its untagged binary messages extensible in the same sense that tagged formats like XML are extensible. When we say that tagged formats are extensible we really mean the following two properties.

1. A tag retains its meaning no matter what other tagged values are present in a message. Therefore, older systems can ignore values it doesn't understand.
2. The absence of a tagged value can be detected. Therefore, newer systems are aware that the value is missing.

Gryphon's schema interpretation tools achieve these goals, essential to allowing message formats to evolve without recompiling applications. As we have chosen to concentrate on performance issues here, the details of how this is achieved is beyond the scope of this paper.

V. Performance Measurements

Sections V-A, V-B and V-C describe performance measurements for the corresponding I/O optimizations discussed in Section III. Section V-D describes performance measurements for the message formatting architecture presented in Section IV. To isolate the impact of I/O optimizations and message formatting architecture, the I/O experiments do not use content filtering, and message format experiments perform no I/O.

All tests were conducted on a local network of twelve IBM F80 6-processor workstations running the AIX operating system and IBM's JDK 1.3. The workstations are connected by both a gigabit and 100 Mbit ethernet LAN. All I/O performance tests were executed on the gigabit network.

A. Cut-thru Performance

The effects of cut-thru were evaluated by fixing the number of publishers (and hence the number of topics) and the message input rate, and varying the number of topic-based subscribers. In particular, six publishers were configured, each publishing on a separate topic, with an aggregate input rate of 150 messages/second (25 msgs/sec/topic). The number of subscribers was varied from 500 to 7000 with subscribers distributed evenly over all available topics (one topic per subscriber). This yields an expected message output rate of 25 messages/second/subscriber. All publishers and subscribers were connected to a single broker. Note that since each publisher publishes on a separate topic, two publishers will never compete for the same subscriber. Figures 2 and 3 summarize the results with cut-thru enabled and disabled. Each test was executed for 30 minutes.

Cut-thru attempts to reduce latency and increase scalability by skipping unnecessary intermediate queuing, and by self-throttling publishers when client load dominates the cost to process a message. This effect is apparent in our sample runs when more than 3000 subscribers are connected. At this point, message delivery begins to dominate the cost of processing an inbound message.

With cut-thru enabled, a single thread is used to both accept and deliver an inbound message to clients. As a result, both inbound and outbound message rates are throttled when delivery cost dominates and the system is overloaded. This is

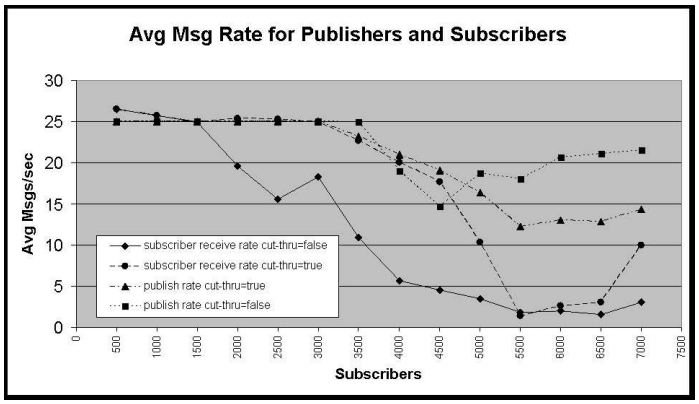

Fig. 2. Average publish rate and subscriber receive rate versus number of subscribers, with cut-thru enabled and disabled.

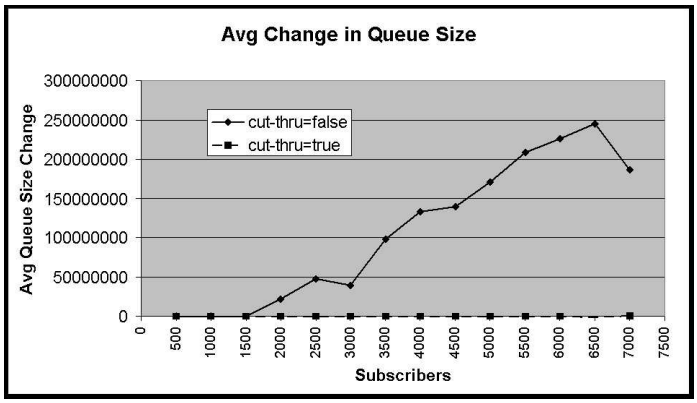

Fig. 3. Average change in the aggregate outbound queue size for subscribers versus number of subscribers, with cut-thru enabled and disabled.

demonstrated by the close correlation between subscriber receive rate and publish rate in Figure 2 when between 3000 and 4500 clients are connected (the system is moderately overloaded in this range).

With cut-thru disabled, publish and subscriber receive rates are relatively independent as long as sufficient buffering resources are available to the system. As a result, publish rate remains relatively steady in Figure 2, whereas subscriber receive rate degrades rapidly as queuing and delivery overhead dominate. As a secondary effect, delivery queues grow monotonically (Figure 3) as delivery threads fall further behind.

From figure 2, we can compare the peak subscriber load that can be supported in the 2 cases, while delivering 25 msgs/sec/topic to each subscriber. We see that cut-thru disabled can support 1500 subscribers, while cut-thru enabled can support 3000 subscribers, a factor of 2 improvement.

B. Socket Hierarchies Performance

The effect of socket hierarchies was evaluated by observing CPU utilization in a mixed environment of relatively few "hot" publishers and subscribers, and a relatively large number of "cold" publishers and subscribers. For each test, a "cold" publisher published on 10 topics at a rate of 0.1 msg/sec/topic. These messages were received by 4000 "cold" subscribers,

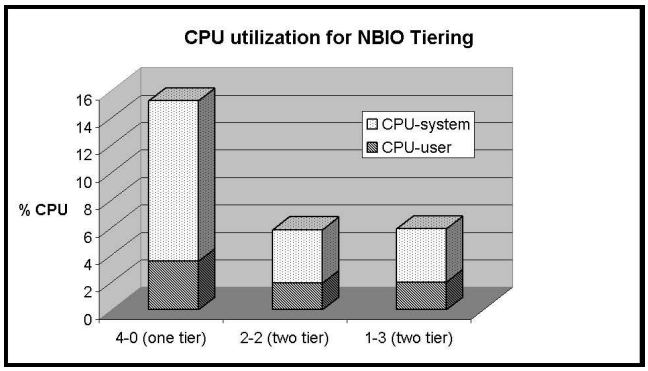

Fig. 4. CPU utilization under one and two tier socket hierarchies. The terminology X-Y indicates a test with X poll lists in tier 1 and Y poll lists in tier 2.

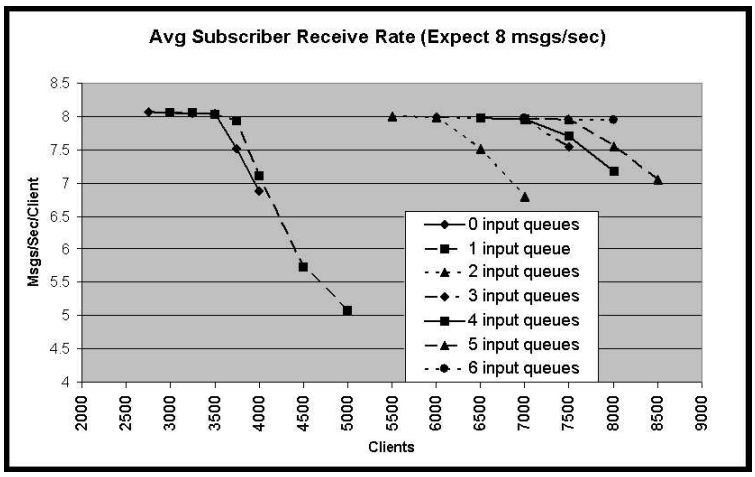

Fig. 5. Average subscriber receive rate versus number of clients and input queues.

which were equally distributed among the "cold" topics (i.e. 400 subscribers/topic). This yields an aggregate rate of 400 "cold" msgs/sec through the broker. Similarly, four "hot" publishers each published on a unique topic at a rate of 20 msgs/sec/topic. These messages were received by 400 "hot" subscribers which were equally distributed among the "hot" topics (i.e. 100 subscribers/topic). This yields an aggregate rate of 8000 "hot" msgs/sec through the broker. Typically, all clients will be equally spread among the available first tier poll lists. Slower clients may migrate to the second tier (if available) as described in Section III.

Three tests were executed with three different socket hierarchy configurations. Cut thru was enabled for all tests and the message rate was not high enough to cause outbound queuing. As a result, most of the poll cycles in the broker were caused by new inbound data from one of the publishers. The results are summarized in Figure 4.

The first test uses a single tier hierarchy with all clients evenly divided among four poll lists. A single tier increases the size of individual poll lists which increases broker overhead each time a poll list is assembled. This behavior is illustrated by the user CPU utilization, where poll lists are manipulated at least as frequently as the "hot" publish rate. Note that system CPU utilization is proportional to the poll list size, and is similarly high in this test.

The second and third tests illustrate the advantages of providing two tiers and allowing slower clients to be handled less frequently. In each of these tests, all clients initially start in the top tier. The lack of queuing on the outbound side[2] causes all subscribers to eventually migrate to the second tier. Similarly, the "cold" publishers are eventually migrated as well. This leaves only the "hot" publishers which are sufficiently active to remain in the first tier. As a result, the size of the poll lists at the top tier are significantly reduced, resulting in lower overall user and system CPU utilization. The second tier poll lists, although much larger, are polled less frequently and contribute little to overall CPU utilization.

[2]Recall that the message rate for these tests is sufficiently low so that cut-thru always succeeds in writing a message directly on an outbound connection. This eliminates the need to poll subscribers.

C. Input Queuing Performance

The effects of input queuing were evaluated by fixing the number of publishers and publisher rate, and varying the number of input queues. For each selection of input queues, clients were added until the system could no longer maintain the desired subscriber receive rate.

The topology for this test consisted of two brokers linked by a single connection. Six publishers were connected to one broker publishing at an aggregate rate of 960 msgs/sec over 120 separate topics. The topics were equally divided among the publishers yielding a message rate of 8 msgs/sec/topic. All subscribers were connected to the second broker, where each subscriber was associated with a single topic. This yields an expected message receive rate of 8 msgs/sec/subscriber. The Mbit ethernet was used for the broker connections and the gigabit ethernet for the client connections.

The number of input queues was varied from zero to six. An upper limit of six was chosen as this would maximize the available concurrency on the 6-processor test machines. In particular, since publishers are evenly distributed among available input queues (within the constraints of message ordering), the six input queue case will dedicate a single input queue to each publisher.

Tests were executed only in the range of clients known to saturate a given input queuing configuration. Figure 5 summarizes the results.

There is little difference between no input queueing and one input queue: all inbound messages will be handled by a single thread. This accounts for the similarity in performance for these two cases.

With two input queues, we see a dramatic improvement in performance with nearly twice as many clients supportable at a steady receive rate. This is not surprising since two input queues will divide the inbound messages equally between two threads servicing clients. Thus, we would expect to be able to support roughly double the number of clients in the zero or one queue case.

With three input queues, we see further improvement, though

not as dramatic as in the two input queue case. Although there is more available concurrency for processing inbound messages, there is also more contention at the system I/O level with each thread competing to write messages to clients. A similar effect occurs when four, five or six input queues are used, resulting in smaller improvements for these cases.

In particular, in the six input queue case, we were not able to proceed further due to excessive congestion on the broker-broker link, which results in buildup of the outgoing message queue on the broker connection at the first broker. Note that this congestion occurs despite the fact that the first broker is not increasing the number of messages it sends to the second broker, and the broker connection is on a different physical link (100Mbit ethernet) from the client traffic (which is increasing). Our investigation eliminated our broker code, and resources managed by the broker code (such as queue sizes) as the cause of this congestion. In addition, the second broker has enough spare CPU resources to read messages at a much faster rate than are being sent by the first broker. We have narrowed the problem to resource contention in the system I/O layer on the machine hosting the second broker. We intend to investigate further to identify the exact cause, and if it can be alleviated by increasing the resource limit settings.

D. Message Format Performance

Message format performance was evaluated using six different message schemas.

1. The **flatInt** schema was as follows:

```
flatInt: [
 field0: int,
 field1: int,
 ...
 field9: int,
 fieldA: int,
 ...
 fieldF: int
]
```

with sixteen fields in all.

2. The **flatString** schema was identical to **flatInt** except all fields were **string** rather than **int**.

3. The **nestedInt** schema organized the same sixteen fields into a four-deep hierarchy. There were two "parts," each with 8 fields, divided into four "sections" with four fields each, divided into eight areas of two fields each, and finally sixteen "pieces" each containing a field.

```
nestedInt: [
  partA: [
    sectionA: [
      areaA: [
        pieceA: [
          field0: int
        ],
        pieceB: [
          field1: int
        ]
      ],
      areaB: [
        pieceC: [
          field2: int
        ],
        ...
        ],
        pieceN: [
          fieldD: int
        ]
      ],
      areaH: [
        pieceO: [
          fieldE: int
        ],
        pieceP: [
          fieldF: int
        ]
      ]
    ]
  ]
]
```

4. The **nestedString** schema was like **nestedInt**, but using **string** values.

5. The **optNestInt** schema was a variation on **nestedInt** in which every fourth field was rendered optional through the use of a variant.

```
optNestInt: [
  partA: [
    sectionA: [
      areaA: [
       ...
      ],
      areaB: [
        pieceC: [
          field2: int
        ],
        { [] |
  pieceD: [
    field3: int
  ]
        }
      ]
    ],
    ...
    sectionD: [
      areaG: [
        ...
      ],
      areaH: [
        pieceO: [
          fieldE: int
        ],
        { [] |
          pieceP: [
            fieldF: int
          ]
        }
      ]
    ]
  ]
```

]
6. The **optNestString** schema was like **optNestInt**, but using **string** values.

We measured the time it took to retrieve a single field from a message (chosen at random) and also the time it took to retrieve nine randomly chosen fields.

We randomly generated 10,000 messages serialized to byte array form. Each message existed in two forms with identical contents. One was Gryphon message format, the other XML (scalar primitive values were expressed as XML attributes and nested structure was expressed by nesting XML elements). Four different samples were used (the results shown below are averages). Variance between samples was negligible.

XML value retrieval was done using the Xerces DOM parser available from apache.org. To avoid unfairly penalizing XML, retrieval of values from the XML message did not start with path expressions which would themselves require parsing but rather with the parsed form of such paths (as arrays of string tag values). The equivalent retrieval tokens for Gryphon message format were integers resulting from numbering the nodes of the schema. In both cases, the appropriate tokens were computed from the schema once only and this computation was not repeated for each message.

The times, in microseconds, to retrieve a single value from each message are shown in the following table.

Schema	Gryphon	XML
flatInt	5.5	948.9
flatString	7.7	948.3
nestedInt	5.5	1306.9
nestedString	7.7	1299.4
optNestInt	5.5	1268.6
optNestString	7.5	1269.1

The times to retrieve nine fields from each message are shown next.

Schema	Gryphon	XML
flatInt	27.5	1000.0
flatString	36.4	982.6
nestedInt	27.6	1693.4
nestedString	36.4	1671.0
optNestInt	25.4	1617.2
optNestString	33.4	1625.6

The most striking feature of the results is that the gryphon format is at least 25 times faster and up to 225 times faster in some cases. Since we used an off-the-shelf DOM parser and did not attempt to do selective parsing with the XML messages, a highly-tuned XML-based implementation will likely to better than what is shown here. But, these results certainly illustrate that attention to message formatting details can produce dramatic results.

A second important thing to note is that gryphon message processing time is quite insensitive to the message layout, at least for the cases shown (recursive schemas and lists of variants were not measured in this benchmark). The results show that Processing time depends only on the type of data retrieved and the number of fields retrieved.

The third observation is that XML processing has a high fixed cost no matter how many fields are accessed while the gryphon processing time is much more sensitive to the number of fields accessed. We believe that this is what you want in a content-based pub/sub system.

Finally, there are some minor additional sources of variation. XML times are generally better with strings than with integers while that difference is reversed for the gryphon format. This is simply a consequence of the scalar encoding formats used in the two cases. Also, both formats run slightly faster when some of the data is missing. This is simply due to the fact that both are able to stop processing when they detect that no value is present in the message.

VI. Related Work

There are several publish/subscribe systems that focus on scalability and low latency, such as Siena [5], Elvin [6], [15], Rebeca [7]. However, none of these have addressed the issue of performing efficient I/O with large numbers of clients or peer brokers with widely differing message rates and capacities. They also do not address the problem of message formatting for fast content matching.

An efficient I/O mechanism is critical to achieve scalability and latency. For example, Elvin [6] has identified threads as a major structural and performance issue for the Elvin implementation. The Elvin server, by its nature, involves large amounts of I/O coupled with fairly intensive computation. Threads were chosen to reduce the latency of notification traffic. This is the same point of view as the Gryphon system. However, Elvin maintained a thread for connection handling, a read, write and notification evaluation thread for each active connection, and a single thread for updating subscription expressions (and emitting quench expressions). This approach does not allow the Elvin server to scale to large number of clients because the maximum number of threads allowed by the underlying system will become a bottleneck. In addition, operating systems usually operate well with a reasonable amount of threads which is usually much less than the maximum number of threads allowed. The overhead with large number of threads, such as cache and TLB misses, scheduling overhead and lock contention, yields a system with suboptimal performance.

What's more, none of the aforementioned systems identifies the large diversity and variation in different client connections, such as a usually-high-input-rate publisher client connection as opposed to a usually-low-input-rate subscriber client connection. And since server to server connections are the backbone of a publish/subscribe network (or server federation), the server to server connections usually aggregate the input from various publishers and thus are of extremely high rate of communication. This makes one thread per connection insufficient.

Java has become more and more important as the language of choice for middleware systems. Besides Gryphon is implemented in Java, Sienna has switched from a C implementation to a Java implementation. Starting from JDK1.4, Java also provides non-blocking I/O operation. Not surprisingly, the JDK1.4 non-blocking I/O [16] is similar to the unix "poll" or "select" function and we can expect implementations using them through Java Native Interfaces. This feature, however, will only solve part of the problem a publish/subscribe system would face, namely, the non-blocking of threads while waiting for socket reads to be available. As a general framework, Java

non-blocking I/O cannot solve the problem with the large number of connections present in a high-volume publish/subscribe system. More specific solutions, such as Gryphon's socket hierarchy schema, are needed to address the situation where connections have high variance in the frequency of communication.

SEDA [17] proposed a design framework for highly concurrent Internet server applications. The proposed architecture is a staged event-driven architecture. The platform provides efficient, scalable I/O interfaces as well as several resource control mechanisms, including thread pool sizing and dynamic event scheduling. In SEDA, applications are divided into stages with queues interconnecting adjacent stages. SEDA's scalable non-blocking I/O mechanism uses a small number of threads per stage, rather than a single thread per task as in Gryphon. Stages thus can run in sequence or in parallel, or a combination of the two, depending upon the characteristic of the thread system and scheduler.

VII. Discussion

In this paper, we have shown that a publish/subscribe messaging system can be built for high performance and near linear scalability. Given these results there are several interesting implications on related research areas. In this section, we outline some of these areas and describe some of the key implications.

- **message processing performance:** As shown in Section V-D, the Gryphon system has achieved very efficient main-line message processing performance. In fact, user-space processing consumes only 40% of a fully loaded broker's cpu capacity, the other 60% being consumed by the O/S. As such, for a broker-based messaging system, there is little room for further optimization of user-space message processing. This work may serve as a point of reference for researchers evaluating performance of new messaging function. Also, it is an interesting area of research on how advanced O/S techniques may be applied to reducing the system-time cost of message processing [18]. In the past, such techniques have been applied extensively to web serving and it is an open question if such techniques could also be effective in a messaging environment.
- **peer-to-peer messaging networks:** A common argument for peer-to-peer messaging networks is their inherent scalability. However, with the scalability of a system such as Gryphon, the need to use peer-to-peer networks for scalability becomes compelling only at the very highest scales (and peer-to-peer systems need to be shown to scale to this degree in practice). As such, the benefits of peer-to-peer messaging is primarily in eliminating the need for a centralized infrastructure. This infrastructure cost must be weighed against the generally increased complexity in the resulting system, the increased load on clients that must also forward messages, and the difficulties in deploying such systems in networks, such as the Internet, dominated by private firewalls.
- **network multicast messaging systems:** The traditional answer to highly scalable messaging systems has been to exploit underlying network multicast protocols. However, these protocols have not been demonstrated to scale well over wide areas [19], and do not map well to very selective but overlapping subscriptions [20]. Gryphon brokers solve these two problems well, with its support for high speed broker-to-broker links and for content-based subscriptions. We view Gryphon message brokering technology as complementary to network multicast-based messaging, and are exploring using multicast for distribution of common data (e.g. financial market pricing data) in a local-area setting at the "edges" of the network while using Gryphon for more selective data and for the long-haul connections between LANs.

References

[1] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting, "Consul: A Communication Substrate for Fault-Tolerant Distributed Programs," Tech. Rep. TR 91-32, Dept. of Computer Science, The University of Arizona, November 1991.

[2] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen, "The Information Bus - An Architecture for Extensible Distributed Systems," *Operating Systems Review*, vol. 27, no. 5, December 1993.

[3] David Powell, "Group Communication," *Communications of the ACM*, vol. 39, no. 4, pp. 50–97, April 1996, (Guest Editor).

[4] Dale Skeen, "Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview," Tech. Rep., Vitria Technology Inc., 1996, http://www.vitria.com.

[5] Antonio Carzaniga, *Architectures for an Event Notification Service Scalable to Wide-area Networks*, Ph.D. thesis, Politecnico di Milano, December 1998.

[6] B. Segall and D. Arnold, "Elvin has left the building: A publish/subscribe notifi cation service with quenching," in *Proceedings of AUUG97*, 1997.

[7] Gero Mühl, *Large-Scale Content-Based Publish/Subscribe Systems*, Ph.D. thesis, Darmstadt University of Technology, September 2002.

[8] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching events in a content-based subscription system," in *Proceedings of the Principles of Distributed Computing, 1999*, May 1999, pp. 53–61.

[9] K. Bruce, R. Di Cosmo, and G. Longo, "Provable isomorphisms of types," *Mathematical Structures in Computer Science*, vol. 2, no. 2, pp. 231–247, 1992.

[10] R. Di Cosmo, *Isomorphisms of Types: from λ-calculus to information retrieval and language design*, Birkhauser, 1995.

[11] M. Rittri, "Using types as search keys in function libraries," *Journal of Functional Programming*, vol. 1, no. 1, pp. 71–89, 1991.

[12] A. M. Zaremski and J. M. Wing, "Signature matching: a tool for using software libraries," *ACM Transactions on Software Engineering Methodology (TOSEM)*, April 1995.

[13] D. J. Barrett, A. Kaplan, and J. C. Wileden, "Automated support for seamless interoperability in polylingual software systems," in *Fourth Symposium on the Foundations of Software Engineering (FOSE)*, October 1996.

[14] Joshua Auerbach, Charles Barton, Mark Chu-Carroll, and Mukund Raghavachari, "Mockingbird: Flexible stub compilation from pairs of declarations," in *1999 IEEE International Conference on Distributed Computing Systems*, July 1999.

[15] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, "Content based routing with elvin4," in *Proceedings of AUUG2K, Canberra, Australia*, April 2000.

[16] "New i/o functionality for java 2 standard edition 1.4," in *http://developer.java.sun.com/developer/technicalArticles/releases/nio*, current as of August 2003.

[17] M. Welsh, D. Culler, and Eric Brewer, "Seda:an architecture for well-conditioned scalable internet services," in *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, 2001.

[18] Philippe Joubert, Robert B. King, Richard Neves, Mark Russinovich, and John M. Tracey, "High-performance memory-based web servers: Kernel and user-space performance," in *USENIX Annual Technical Conference*, 2001, pp. 175–187.

[19] Kenneth Birman, Andre Schiper, and Pat Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems*, vol. 9, no. 3, pp. 272–314, August 1991.

[20] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, "An effi cient multicast protocol for content-based publish-subscribe systems," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999*, 1999, pp. 262–272.