

Reflection in the Gryphon Message Brokering System

Daniel Sturman

Guruduth Banavar

Robert Strom

IBM T J. Watson Research Center

30 Saw Mill River Rd.

Hawthorne, NY 10532

{ sturman | banavar | strom }@watson.ibm.com

ABSTRACT

Gryphon is a new paradigm for *message brokering* which attempts to merge the best features of distributed publish/subscribe communications technology and

A Gryphon information flow graph is an abstraction because Gryphon is free to physically implement the flow any way it chooses. Implementation may radically alter the flow pattern, provided that the consumers see the appropriate streams of events consistent with the delivered events and the transformations specified by the graph. Gryphon optimizes graphs and deploys them over a network of *brokers*. The broker network is responsible for handling client connections and for distributing events.

Information Spaces (nodes)

Each node in the graph is called an *information space*. Each information space has a schema called its *type*

TYPE	MEANING	TYPE SYNTAX	EXAMPLE
atom	uninterpreted data	atom	image
string	an identifier	string	"Gryphon"
integer	a count	int	30
bag	unordered collection of elements of a given type	{ type }	{"Gryphon", "Event"}
list	ordered collection of elements of a given type	(type)	(["IBM",80],["HP",20])
tuple	collection of fields	[[id:] type [, [id:] type] ...]	["IBM",80]
variant	value with tag-dependent format	union [tag:type[,tag:type] ...]	Sell: ["IBM",80]

Table 1: Gryphon Event Types

Domains

Information spaces are organized into *domains* to provide structure and security boundaries. System services are often specialized for particular domains. In particular, each domain has an autonomous authorization policy. Domains may include part of a single information flow graph or may contain multiple graphs. A single information space, however, may only belong to a single domain, and domains may not contain each other or overlap in any way.

Filter & Transformation Language

Events are categorized by their *type*, which defines the structure and content of an event (Table 1). Atoms, strings and integers are organized into structures by collecting them in *bags*, *lists*, or *tuples*. A bag is an *unordered* collection of elements, while a list is an *ordered* collection. The elements stored in bags and lists must all be of the same type. A tuple is a special type of list (*i.e.* ordered collection) in which the size is fixed and each element may have a different, fixed type. A *variant* is a type which consists of a *selector* that distinguishes among one or more sub-types. Variant selectors are specified as a collection of tags, each of which is followed by a type.

Table 2 shows a subset of the Gryphon pattern matching language. Select arcs are annotated with a single pattern describing those events passed by the arc, *i.e.* the select

filter function. Transformation arcs describe a transformation function in the form:

$$Pattern \Rightarrow Pattern$$

where the variables bound in the first pattern may be referenced in the second. For example, in Figure 1, events containing a "price" and "volume" field are transformed into events where these fields are replaced by a "capital" field.

META-SPACES

For each domain there exists a *meta-space*. This meta-space is a single information space which contains events representing changes to information flow graphs in a domain. Some events may affect multiple domains such as when an arc joins information spaces in two different domains. In this case, the event appears in both meta-spaces. Other information spaces may be derived from a meta-space using merge, select, and transform arcs. Each event in the meta-space includes the name of the principal on whose behalf the action is performed and the name of the broker to which the principal is connected.

The following meta-level events are generated by the system and placed in the meta-space:

- addArc - An arc is added to the information flow graph. This event includes an arc name, the head and tail of the arc, and description of the label on the arc. System components interested in additions to the information

PATTERN SYNTAX	MATCHES	EXAMPLE
constant	its value	"a string"
bound variable		30
var [range]	any value in range; binds var	price > 1000
tag: pattern	variant satisfying pattern	Sell: [issue: "IBM"]
[pattern [, pattern] ...]	tuple satisfying all patterns	["IBM", price > 75, any]
[id: pattern [, id: pattern] ...]		[price: p > 75]
constant + pattern	bag containing the constant or bound variable and whose	{[issue:"IBM",price:80]}+rest
bound-variable + pattern	remaining elements match the pattern	

Table 2: Filter & Transformation Pattern Syntax

flow graph may subscribe to addArc events, and those components that are authorized to modify the graph may publish these events. More formally, this event is of the type:

```
[arcName: string, srcNode: string, destNode: string, func: union [
  Transform: [ srcPat: string, resultPat: string ] ,
  Select : [ pat: string ],
  Merge: []]
```

For addArc events with a srcNode and destNode in different domains, an identical event must be published in each domain's meta-space for the arc to be created.

- deleteArc - An arc is removed from the information flow graph. This event includes the name of the arc to be deleted. System components interested in removals from the information flow graph may subscribe to deleteArc events, and those components that are authorized to modify the graph may publish these events. This event is of the type:

```
[ arcName: string]
```

- addSpace - An information space is added to the information flow graph (initially without any edges). These events may include adding a new client to the system.

```
[ nodeName: string]
```

- deleteSpace - An information space is removed from the information flow graph. These events include the exit of a client from the system.

```
[nodeName: string]
```

- requestAddArc - Most clients and services in a system will not be authorized to arbitrarily modify the information flow graph. These components instead issue modification requests that are acted on by authorized system services. Requesters may know when a request has been granted by listening for addArc or requestProcessed events. These events have a type similar to an addArc event, but include a request identifier:

```
[ id: int, arcName: string, srcNode: string, destNode: string, func: union [
  Transform: [ srcPat: string, resultPat: string ] ,
  Select : [ pat: string ],
  Merge: []]
```

- requestDeleteArc - This request is issued by components wishing to delete an arc. These events have a type similar to a deleteArc event, but include a request identifier:

```
[ id: int, arcName: string]
```

- requestProcessed - When requests are accepted or denied, events of this type are issued. These events are not really meta-level events in that they do not represent underlying system behavior, but are provided in the meta-space to allow arc requesters to listen for a response to their request in a single information space. These events have the type:

```
[ id: int, success: boolean, reason: string]
```

- disableArc - To handle disconnecting clients and to provide a degree of failure notification, events of this type

are issued. A disabled arc is one that has not been deleted, but from which events will not flow until the arc is enabled. These events have the type:

```
[arcName: string]
```

- enableArc - This event corresponds to the enablement of a previously disabled arc. These events have the type:

```
[arcName: string]
```

Together, the above nine event types define the type of the meta-space for information flow graphs. Thus, the formal type of the meta-space is:

```
[ principal: string, broker: string, action: union [
  AddArc: [ arcName: string, srcNode: string, destNode: string, func: union [
    Transform: [ srcPat: string, resultPat: string ] ,
    Select : [ pat: string ],
    Merge: []]
  DeleteArc: [ arcName: string],
  AddSpace: [ nodeName: string],
  DeleteSpace: [ nodeName: string],
  RequestAddArc: [ id: int, arcName: string, srcNode: string,
  destNode: string, func: union [
    Transform: [ srcPat: string, resultPat: string ] ,
    Select : [ pat: string ],
    Merge: []],
  RequestDeleteArc: [ id: int, arcName: string],
  RequestProcessed: [ id: int, success: boolean, reason: string ],
  DisableArc: [ arcName: string],
  EnableArc: [ arcName: string]]]
```

APPLICATIONS OF META-SPACES

We illustrate two classes of problems that may be solved by using a meta-space. The first involves deriving a service to provide clients with awareness of system characteristics. The second involves creating new system services that interpret system requests.

In this section we provide an example in each class. For the first class, we define a location monitor space which publishes events when clients of the system change (physical) location. This service adds value for clients but does not modify system behavior in any way. For the second class, we demonstrate how a message warehousing service may be integrated seamlessly into the system.

Location Monitor Service

The location monitor information space is defined using the information flow graph shown in Figure 2. The second arc on the EnableArc side of the diagram requires some explanation. This transform is known as a *join transform*. Join transforms are a special case where messages are compared against a database (assumed to be static) in the same manner as joins between databases are performed. In this case, the event is joined against the database "brokerLocations", creating a new event with the client's name and its current physical location as defined by the broker to which it connected. Note that for this example, it is not even necessary to explicitly run a service: the entire Location information space may be defined using the selects, transform, and merge functionality provided in information flow graphs.

Message Warehousing Service

Our second example requires that a service process be created. To support message warehousing requires a warehouse process that will both subscribe to and publish meta-events. The information flow for this example is shown in Figure 3.

The warehouse monitors the creation of new information spaces. Upon receiving an AddSpace event, the warehouse adds an arc from this event source to itself. The warehouse service also subscribes to requests by clients for connectivity into the graph. Upon receiving an ArcRequest event, the warehouse generates a request of its own to connect itself to the new client making the request. This connection to the client involves a select arc selecting only those message from the warehouse destined for the client, and a transform stripping this destination information from the message. Note that these two events are requests. It would be possible to simply add these arcs, but by making them request, we allow the system security managers to control warehousing.

REFERENCES

1. K. P. Birman. "The process group approach to reliable distributed computing," pages 36-53, *Communications of the ACM*, Vol. 36, No. 12, Dec. 1993.
2. P. Maes. *Computational Reflection*. Technical Report 87-2, Artificial Intelligence Laboratory, Vrije University, 1987.
3. Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. *Consul: A Communication Substrate for Fault-Tolerant Distributed Programs*, Dept. of computer science, The University of Arizona, TR 91-32, Nov. 1991.
4. Brian Oki, Manfred Pfluegl, Alex Siegel, Dale Skeen. "The Information Bus - An Architecture for Extensible Distributed Systems," pages 58-68, *Operating Systems Review*, Vol. 27, No. 5, Dec. 1993.
5. David Powell (Guest editor). "Group Communication", pages 50-97, *Communications of the ACM*, Vol. 39, No. 4, April 1996.
6. Dale Skeen. *Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview*, <http://www.vitria.com/>
7. B. C. Smith. *Reflection and Semantics in a Procedural Language*. Technical Report 272, Massachusetts Institute of Technology. Laboratory for Computer Science, 1982.
8. Rob Strom, Daniel Sturman, Mark Astley, Tushar Chandra. *Scalable Matching and Multicast for Content-Based Subscription*, Unpublished.