Exposing the network: Support for topology-sensitive applications

Youngsu Chae † Shashidhar Merugu † Ellen W. Zegura † Samrat Bhattacharjee ‡

† Networking and Telecommunications Group
College of Computing, Georgia Tech, Atlanta, GA
{yschae,merugu,ewz}@cc.gatech.edu

‡ Department of Computer Science
University of Maryland, College Park, MD
bobby@cs.umd.edu

Abstract—**One of the traditional goals of networking has been to hide details of network topology from end users. As networks become larger and more heterogeneous, however, situations arise in which the ability to identify particular topological properties enables capabilities and performance that are difficult to achieve with a purely "black box" interface to network topology. Examples of such situations include deployment of active networking functionality to strategic points (e.g., upstream from a lossy link) or the construction of a secure overlay topology on a network with selective support for IP security. On the other hand, an approach that "opens up" the network without constraints seems neither necessary nor practical. We therefore propose a method to query and synthesize network information that allows constrained programmability. We demonstrate the method on a set of examples, and we discuss our implementation within an active networking environment.**

Keywords—**active networks, programming interfaces, active services**

I. Introduction

One of the traditional goals of networking has been to hide details of network topology from end users. For example, a sender can use IP to transmit packets to any destination, without regard to (or information about) the topology between the source and destination, beyond connectivity[1]. As networks become larger and more heterogeneous, however, situations arise in which the ability to identify particular topological properties enables capabilities and performance that are difficult to achieve with a purely "black box" interface to network topology.

For example, consider the following problems:

- Determine the capacity on the most-constrained link in the multicast tree rooted at a particular source host. This information could be used to affect the source encoding of the data so that it is suitable for the least capable receiver.
- Place a repair server at the upstream end of any link in the multicast tree from source A to multicast receiver group R, whose loss rate exceeds p packets per second. The purpose of the repair server is to cache packets from the source and reply to negative acknowledgements in a more timely manner and with less overhead than relying upon the source [1].
- Construct a virtual topology whose links and nodes (i.e., routers and hosts) are secure based on support for IPsec [2] and an adequate distribution of shared keys.

Access to internal network topology information is critical to these problems; hiding topology makes them difficult or impossible to solve. On the other hand, an approach that "opens up" the network without constraints seems neither necessary nor practical. The examples above do not require a fully programmable network interface, with the associated well-known concerns regarding performance and security.

We propose a programmable gather-compute-scatter (GCS) mechanism to query and synthesize network information. The user controls the activity during each phase (gather, compute, scatter) and can prescribe multiple iterations. However, the compute phase is restricted in access and computation, balancing programmability with performance and security. The primary advantages over traditional MIB queries are (1) dynamic control over where queries are performed, (2) network-embedded synthesis of results, and (3) dynamic control over where results are sent. These advantages translate into savings in bandwidth and time over a centralized and non-programmable query system.

The paper is organized as follows. The next section states our assumptions about the network and describes the GCS mechanism in some detail. In Section III we illustrate the use of the mechanism through a set of examples. Section IV describes our implementation within an active networking environment and reports on a performance experiment. Related work is described in Section V, and we conclude in Section VI.

II. The Gather-Compute-Scatter Mechanism

A. Network model

We assume a network consisting of *nodes* (i.e., routers or hosts) and *links* (i.e., channels on which nodes can send messages). Links may be physical transmission media, or they may be a higher layer entity. Both nodes and links have *attributes* providing static or dynamic information. For example, an attribute might be the average queue length for a link or the support for a particular protocol at a node. We assume that each node has a local interface that can be used to read the attribute values for the node and any incident links. Each node and link has an attribute which is a globally unique identifier. In addition to the attribute storage, we assume each node has a state store that can be used by applications to hold temporary information.

B. IGCS overview

We propose the Iterative Gather-Compute-Scatter (IGCS) distributed computation model to query and synthesize network state. IGCS programs repeat a gather, compute, scatter cycle until a given condition is satisfied. For many programs, the number of iterations is pre-determined and fixed, though this is

Work supported by DARPA under contract number N66001-97-C-8512.

[1]Due to the best-effort nature of IP, of course, these transmitted packets may or may not actually be received.

not required. During an iteration, a set of messages are collected during the gather phase. Once a specific set of messages have been collected, the compute phase commences. The inputs to the computation are the set of collected messages, the node and link attributes, and the state store for this computation. The compute phase can produce a single message (of a fixed IGCS message type) that is then "scattered", i.e. transmitted to a set of destinations. IGCS programs can retain state at a node while they are active. All state is lost after the last iteration of the IGCS computation.

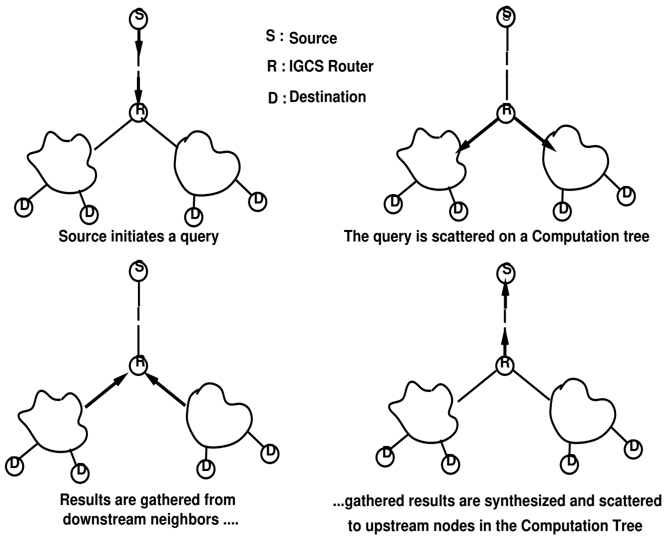


Fig. 1. Snapshots of a distributed computation model to query network state

Figure 1 illustrates four snapshots of this mechanism inside the network, comprising two iterations. The initial scatter phase is used to disseminate a query down a path (or a tree), and the second gather phase is used to accumulate results. The compute phase synthesizes information at each node. We can see how network information (e.g., loss rate) can be gathered up a computation tree (e.g., corresponding to a multicast distribution tree). The gathered information can be processed at each node (e.g., the maximum of all the gathered loss rates is computed) and then forwarded up the computation tree to satisfy a network query.

C. IGCS specification

An IGCS node-level computation is specified by a set of three-tuples: $\{(G_i,\ C_i,\ S_i)\ \mid\ 0 \leq i \leq (n-1)\}$ where n denotes the total number of iterations. The sets G_i and S_i and the function description C_i define the processing during the gather, scatter, and compute phases of the i^{th} iteration, respectively.

The gather phase of the i^{th} iteration is specified by the set G_i, of the form $\{l_1, l_2, \ldots, l_g\}$, where each l_i identifies a specific link incident to the node where the computation is running. Any IGCS message that arrives on one of these links and contains the identifier for this IGCS instantiation will be delivered and stored until one message has been received on each link.

The compute phase of the i^{th} iteration is specified by the function C_i. Any function that can be executed during a compute phase has the following signature:

$$oMsg \leftarrow C\ (\{iMsg\}, \Sigma_{node}, \Sigma_{link}, \Pi)$$

where $oMsg$ is an IGCS message, Π is the state store associated with this node-level computation, $\{iMsg\}$ is the set of messages gathered during the gather phase, Σ_{node} is the current node attributes, and Σ_{link} is the current link attributes. $oMsg$ may be null, corresponding to a function that does not produce an output message.

Thus, functions in the compute phase take as input the state of the node and links, set of messages gathered in the current iteration, and a computation-specific state store. They may produce a single message that is then forwarded in the scatter phase. State accumulated by iterations of a computation is stored in its state store. The state is discarded after the last iteration. As will be illustrated later, one common purpose of the compute phase (in addition to producing an output message) is to generate the gather and scatter sets for subsequent iterations.

The scatter phase in the i^{th} iteration is defined by the set S_i. Each S_i is a set of link descriptors $\{l_1, l_2, \ldots, l_s\}$. During the i^{th} iteration, the message produced by the i^{th} compute phase is forwarded on all links specified by S_i.

D. IGCS message format

The format of an IGCS message is shown in Table I. Each

Key	Value
compId	*Computation Identifier*
srcId	*Source of IGCS Message*
topId	*Topology Identifier*
key	*value*
⋮	⋮
key	*value*

TABLE I
GENERIC IGCS MESSAGE FORMAT

message contains a set of (*key* , *value*) pairs [2]. The first three keys are mandatory in each message. The *compId* key contains a globally unique identifier for the IGCS computation. The computation id is used to demultiplex incoming messages to appropriate IGCS node-computations. The *srcId* key identifies the previous-hop IGCS node that scattered this message. The *topId* key is the unique identifier of the topology over which the IGCS computation is being executed. Along with the mandatory keys, there are a set of optional (*key* , *value*) pairs that may be encoded in each message. There are two types of optional keys: (1) keys that specify the phases and iterations of an IGCS computation (i.e., G_i, C_i, S_i), and (2) keys defined by computations to exchange data (e.g., attributes such as queue length).

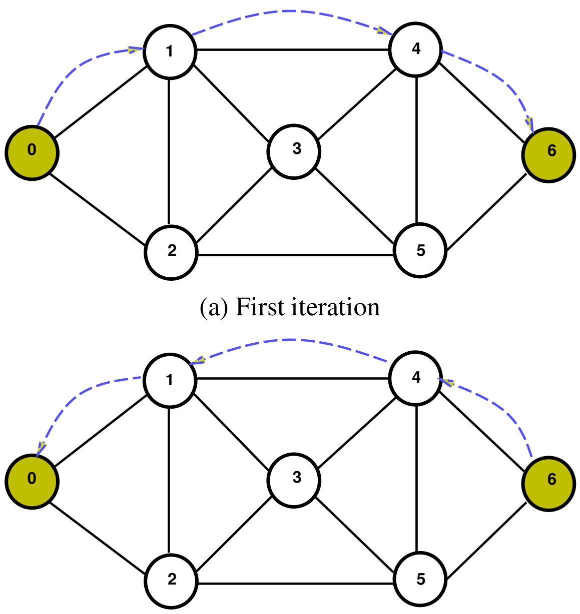
[2] In practice, a mechanism to identify the length of each value is also required. For ease of exposition, we do not address the issue of the length of each field. We initially assume that the underlying topology is able to reliably transmit messages of arbitrary length.

III. Examples of IGCS Computations

To demonstrate how the IGCS computation model works, we describe a few examples. We begin by considering a query example, where the goal is to identify the least available bandwidth on the path from a source to a destination. We then show how a trivial modification allows the IGCS computation to be extended to identify the least available bandwidth on the multicast tree from a source. We then turn to two examples that involve identifying topologies. The first identifies a spanning tree on an underlying topology; the second identifies secure links, making use of the ability to identify a spanning tree.

The examples assume the existence of a function `NextHop(topology,node)` that returns the appropriate link on a given topology to a given destination node. We assume the function returns Null at the destination. The examples also use a function `GetNodeID` that returns the node identifier attribute.

A. Path information retrieval



(a) First iteration

(b) Second iteration

Fig. 2. Least bandwidth finding on path 0-1-4-6

As a first example, consider the problem of finding the least available bandwidth along the path from Node 0 to Node 6 in Figure 2(a). The problem can be solved with an IGCS computation with two iterations. During the first iteration, the IGCS computation is dispersed along the path 0-1-4-6 as shown in Figure 2(a). Figure 2(b) shows message flow in the second iteration, in which the least available bandwidth information is calculated and returned to the source.

The IGCS message for the computation is shown in Figure 3(a). In addition to the required keys, this message also specifies the number of iterations, the initial gather set, the compute functions for the two iterations, and the source and destination nodes for the path. Though the IGCS computation executes for two iterations, the message does not specify S_0, G_1, and S_1. Instead, they are determined during the compute phase of the first iteration.

The first compute phase, shown in Figure 3(b), accomplishes two tasks. First, it distributes the IGCS computation on the path by updating the original message $sMsg$ to reflect the proper $srcId$ and then specifying a scatter to the next node in the path to *dstNode*. Second, the compute phase determines the gather and scatter sets for the next iteration. The second iteration will proceed from the destination back to the source along the reverse path, thus the gather set for the second iteration is exactly the scatter set for the first iteration. The scatter set for the second iteration is the link on which the initial message arrived. (We determine this using the `Nexthop` function and assuming symmetric paths.) Note that at the destination, G_1 will be Null, thus the gather phase of iteration 1 at the destination will (trivially) complete as soon as iteration 0 is done.

During the second compute phase, the information gathered from downstream links is synthesized in the function `cal-min-bw`, shown in Figure 3(c). The local link attributes are used in this function, along with the gathered results stored in $iMsg$, to find the least available bandwidth so far along the path from the current node to the destination. This result is stored as a *(bw,least-value)* pair in message $oMsg$ and sent to the previous hop on the path in the second scatter phase.

Key	Value
$compId$	Unique ID
$srcId$	Node 0
$topId$	T
$numIter$	2
G_0	Null
C_0	`fwd-sig`
C_1	`cal-min-bw`
$srcNode$	Node 0
$dstNode$	Node 6

(a) IGCS Message

`fwd-sig :`		
$oMsg$	$\leftarrow$	$sMsg$
$oMsg.srcId$	$\leftarrow$	`GetNodeId()`
S_0	$\leftarrow$	`NextHop`$(sMsg.topId, sMsg.dstNode)$
G_1	$\leftarrow$	S_0
S_1	$\leftarrow$	`NextHop`$(sMsg.topId, sMsg.srcId)$

(b) Code for C_0

`cal-min-bw :`		
$oMsg$	$\leftarrow$	$sMsg$
$oMsg.srcId$	$\leftarrow$	`GetNodeId()`
$oMsg.bw$	$\leftarrow$	$\min(iMsg_j.bw, link_k.bw, \forall k, j)$

(c) Code for C_1

Fig. 3. IGCS computation for retrieving path property

Extensions

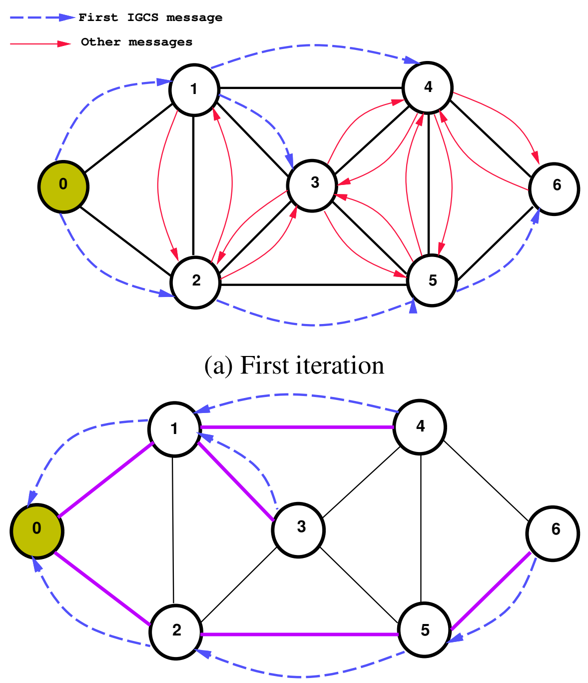
The same IGCS computation can be used to calculate the minimum available bandwidth along a multicast tree. Consider that nodes 0, 3, 5 and 6 are currently participating in a multicast group with address *grAddr*. Assume that Node 0 is a source interested in finding the minimum bandwidth along the multicast tree. We can easily solve the problem by instantiating the same IGCS computation with *grAddr* as *dstNode*.

The current IGCS computation can be easily modified to solve the multicast repair server problem discussed in the introduction. This is achieved by providing a new function for C_1. Instead of gathering minimum bandwidth information, the new function checks whether any of the down-stream links on the multicast tree experience packet loss larger than a certain threshold value. If the loss rate at a node exceeds the threshold, it is added to a list of possible repair locations in the IGCS message. When the computation is done, the multicast source node has the list of candidates and can determine the location(s) of the repair server(s) based its policy.

Comparison with a centralized algorithm

We compare the above IGCS computation with a centralized algorithm in Section IV-C. Briefly, the centralized algorithm requires more time, since it must query each node about the next hop, and results in greater message traffic. We quantify these effects when we compare implementations of both schemes.

B. Building a spanning tree



(a) First iteration

(b) Second iteration

Fig. 4. Building a spanning tree

As the second example, we consider the problem of building a spanning tree topology over a network. A spanning tree is useful for a variety of queries that require network-wide information. We will show in the next example how the spanning tree is used while building a secure sub-topology over a base topology.

We solve the problem with a two-iteration IGCS computation that is based on constrained flooding [3]. During the first iteration, the IGCS computation is flooded over the entire network as in Figure 4(a). Only the first signaling message received at a node initiates a new node-level computation. At each node, the signaling message is flooded onto all the links except the one where the first message received. The link where the first message was received defines a parent-child relationship and forms a spanning tree on the network. During the second iteration, in Figure 4(b), the local spanning tree information is combined with the information received from child nodes and then forwarded along the spanning tree towards the source.

The initial IGCS message for the computation is shown in Figure 5(a). It includes a tree key that is used to return the spanning tree during the second iteration. The computation for the first iteration is shown in Figure 5(b). During the first iteration, we flood the IGCS message by setting S_0 to be all the links except the link from which the IGCS message has been received. Note that because each node executes the first iteration exactly once, the flooding will automatically terminate without looping. As in the previous example, G_1 is set to all the links of S_0, and S_1 is the previous hop for the initial message. The second iteration gathers the sub-tree information from the child node and builds up a new sub-tree that includes the local node. The computation for C_1 is shown in Figure 5(c).

Key	Value
$compId$	Unique ID
$srcId$	Node 0
$topId$	T
$numIter$	2
G_0	Null
C_0	`fwd-flooding`
C_1	`build-sub-tree`
$tree$	Null

(a) IGCS Message

`fwd-flooding :`		
$oMsg$	$\leftarrow$	$sMsg$
$oMsg.srcId$	$\leftarrow$	`GetNodeId()`
S_1	$\leftarrow$	`NextHop`$(sMsg.topId, sMsg.srcId)$
S_0	$\leftarrow$	$\{link_i \in \Sigma_{link}\ ,\ \forall\, i\} - S_1$
G_1	$\leftarrow$	S_0

(b) Code for C_0

`build-sub-tree :`		
$oMsg$	$\leftarrow$	$sMsg$
$oMsg.srcId$	$\leftarrow$	`GetNodeId()`
$TreeSet$	$\leftarrow$	$\bigcup \{iMsg_j.tree\},\ \forall j$
$oMsg.tree$	$\leftarrow$	$\bigcup (TreeSet, S_1)$

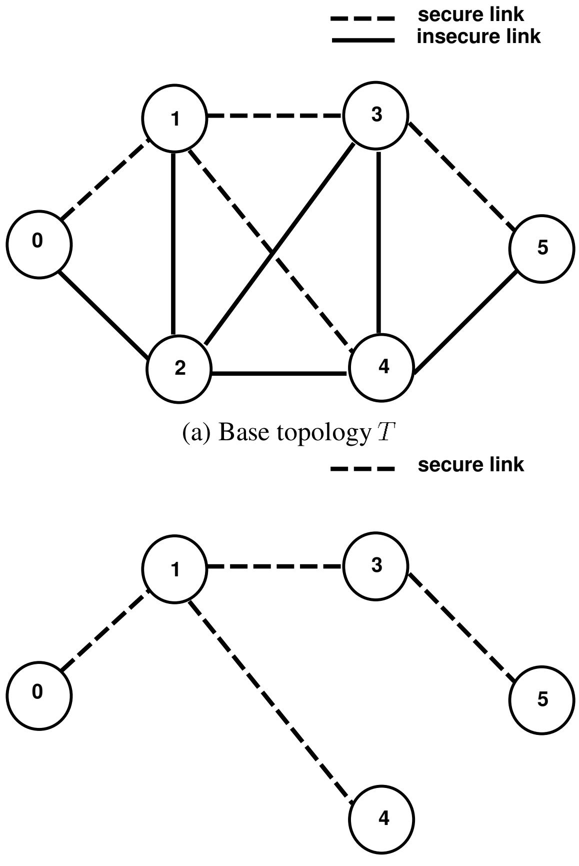
(c) Code for C_1

Fig. 5. IGCS computation for building a spanning tree

Comparison with a centralized algorithm

A centralized algorithm sends out query messages to all the nodes in the network. The reply messages contain the local adjacency information for the replying nodes. The algorithm then combines all the local topology information and builds a spanning tree using a spanning tree finding algorithm such as Dijkstra's. Assume that the network has N nodes and M links. The total number of messages exchanged in the centralized algorithm is $2(N-1)$. For the IGCS computation, two messages are exchanged along each link in the spanning tree. Other links carry one message. Those two approaches result in the same number of messages if the underlying network is a tree topology. As M increases, the IGCS computation results in more message exchanges. However, the maximum number of message exchanged per link is two. Further, the initiating node exchanges only $2d$ messages for d outgoing links. In the centralized algorithm, however, the initiating node exchanges all $2(N-1)$ messages, potential leading to message implosion.

C. Identifying a secure topology



(a) Base topology T

(b) Induced secure topology T_{sec}

Fig. 6. Property-based topology identification

The final example is to identify a new topology, which consists of only secure links, from a base topology. In Figure 6(a), the topology $T=(V,E)$ contains both secure and insecure links. The problem is to build a new topology $T_{sec}=(V_{sec},E_{sec})$ that consists of only secure links as in Figure 6(b).

A two-iteration IGCS computation will solve this problem, if we assume a spanning tree ST has already been computed. The previous example shows how this can be done. The first iteration disperses the IGCS computation on the spanning tree ST. The second iteration retrieves local secure link information and also gathers the secure link information from the child nodes. The set of secure links are scattered to the parent node in the spanning tree. When the computation is done, the root node has the complete set of secure links and can instantiate the secure topology.

Figure 7 shows the IGCS message and the computations for C_0 and C_1. Note that we assume that the `GetChildLink` and the `GetParentLink` functions are given. The `GetChildLink` function identifies the links to the children of the local node on the given spanning tree ST. `GetParentLink` identifies the link to the parent.

For comparison with a centralized algorithm, the topology identification problem is similar to the spanning tree problem. The IGCS computation would have an additional advantage if a distributed topology instantiation scheme were in place, since this would allow IGCS to avoid collecting the topology information in a central location. Distributed topology instantiation is an area for future work.

Key	Value
$compId$	Unique ID
$srcId$	Node 0
$topId$	T
$numIter$	2
G_0	Null
C_0	`fwd-sig-st`
C_1	`build-sub-topo`
$srcNode$	Node 0
$spTree$	ST

(a) IGCS Message

`fwd-sig-st :`		
$oMsg$	$\leftarrow$	$sMsg$
$oMsg.srcId$	$\leftarrow$	`GetNodeId()`
S_0	$\leftarrow$	`GetChildLink(`$sMsg.spTree$`, GetNodeId())`
G_1	$\leftarrow$	S_0
S_1	$\leftarrow$	`GetParentLink(`$sMsg.spTree$`, GetNodeId())`

(b) Code for C_0

`build-sub-topo :`		
$oMsg$	$\leftarrow$	$sMsg$
$oMsg.srcId$	$\leftarrow$	`GetNodeId()`
$LinkSet$	$\leftarrow$	$\{\, link_i \mid link_i.secure = True\ , \forall i \,\}$
$oMsg.link$	$\leftarrow$	$LinkSet \bigcup \{\, link_k \mid link_k \in iMsg1_j.link, \forall j, k \,\}$

(c) Code for C_1

Fig. 7. IGCS computation for building secure topology

IV. IGCS System Implementation

We have implemented IGCS within the CANEs execution environment on top of the Bowman NodeOS [4]. This section provides an overview of the implementation and the results of a performance experiment.

A. *CANEs and Bowman*

CANEs is an execution environment that provides a composable active networking environment. Composition in CANEs is achieved in two steps. First, the user selects an *underlying program*. The underlying program exports one or more *processing slots* that identify the specific points where *injected programs* are bound and executed. Users can select or provide a set of injected programs that can be used to customize the underlying program. All IGCS computations share the same underlying program (described below) that contains only one slot for the compute phase.

Bowman is a NodeOS for active networks. Bowman provides three key resource abstractions: *channels* that are communication end-points, *a-flows* that are the primary abstractions for computation, and *state-store* that provides a mechanism for a-flows to store and retrieve state. The a-flows are used to implement the IGCS node-level computations; channels provide a communication mechanism through which the IGCS node-level computations can exchange messages. The Bowman also provides an efficient packet classification mechanism used to specify the gather and scatter sets.

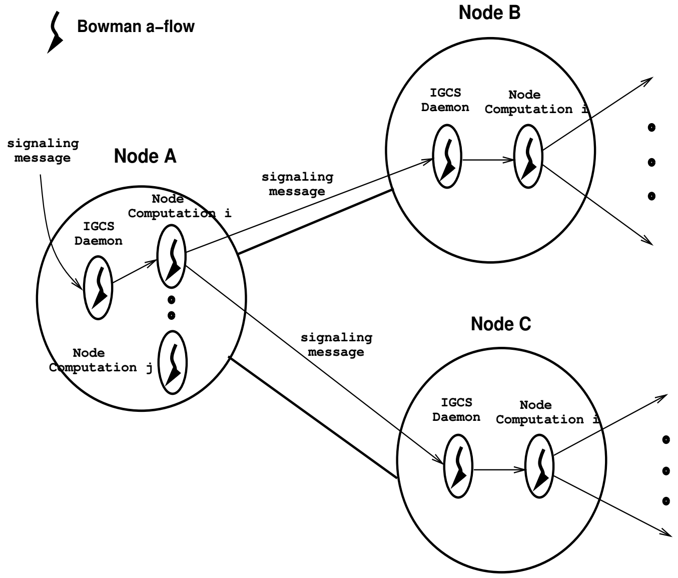


Fig. 8. IGCS system architecture

B. *IGCS system architecture*

The IGCS system has three major components: the IGCS daemon, the IGCS underlying program and the IGCS compute slot program. We describe each in turn.

IGCS daemon : The IGCS daemon is a node-resident program, which is responsible for processing IGCS signaling messages. Upon receiving a signaling message, it parses the message and initiates a node-level computation with a proper set of code modules for the IGCS underlying program and compute slot programs.

The IGCS daemon is implemented as an extension of the Bowman NodeOS. Upon receiving the signaling messages, the daemon configures a new IGCS computation and initiates the new node-level computation using a Bowman a-flow. Figure 8 shows a typical snapshot of the IGCS system. The IGCS daemon on Node A receives a signaling message from the user and initiates a new node-level computation. During the first iteration, the node-level computation computes a set of nodes on which the current computation should be initiated (Nodes B and C in this example) and sends the signaling message to the nodes. Upon receiving the signaling message, the daemons on nodes B and C perform the same operations and spread out the computation.

During the processing, the underlying and compute slot programs are loaded onto the local node via Bowman code loading mechanism. The compute slot programs are bound to the proper compute phases. The data part of the signaling message is stored into the local state-store so that the information can be retrieved by the IGCS computation.

IGCS underlying program : The underlying program provides the computation framework for one or more iterations of the Gather-Compute-Scatter cycle. The underlying program gathers a set of messages through the channels specified in the current gather phase. Then it executes the compute slot program of the current iteration. In the scatter phase, it sends out the messages through the channels specified in the current scatter phase.

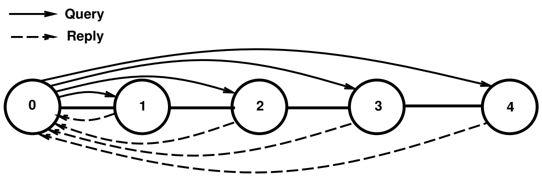
Figure 11 in the Appendix shows the main body of the IGCS underlying program. At each iteration, the underlying program gathers messages by subscribing to the data packets with its own computation id on the gather channels. After gathering all the messages, it stores them into the local state-store for further processing during the compute phase. Next, the underlying program raises the compute slot program which is bound to the current iteration of the compute phase. The "raised" injected program does the computation-specific processing of the current iteration using the gathered messages and local information. It also generates outgoing messages and specifies a set of channels for the scatter phase. The messages are sent out over the scatter channels during the scatter phase.

IGCS Compute slot program : The IGCS compute slot program defines each IGCS computation. The compute slot programs are bound to the proper compute phases and executed during the specified iteration. The slot programs can be specified statically in the signaling message or dynamically modified at run-time.

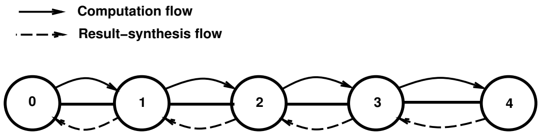
Since all the IGCS computations share the same underlying program, each IGCS computation is specialized through the injected programs that are bound to the compute slots. The IGCS system provides an API through which the slot programs can communicate with the IGCS underlying program and other slot programs bound to the different iterations.

Figure 12 in the Appendix shows a slot program for the IGCS in-band signaling. As in the example, the IGCS API provides a set of functions to retrieve/configure gather/scatter phases and other shared objects. The number of iterations for the IGCS computation and slot programs bound to the compute phases can also be modified "on-the-fly" within compute slot programs. The capability of run-time modification allows dynamic configuration of the IGCS computations.

C. Performance analysis



(a) Centralized algorithm



(b) IGCS algorithm

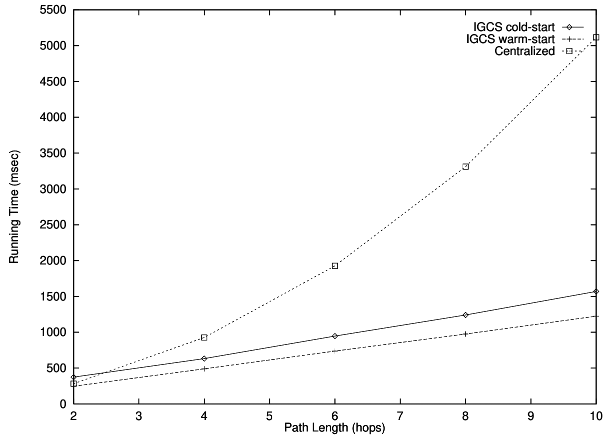
Fig. 9. Finding the minimum available bandwidth

In this section, we present a limited set of IGCS performance results. We have implemented two solutions for finding the minimum available bandwidth: a centralized algorithm and the IGCS algorithm described in Section III. In short, the IGCS algorithm spreads the computation along the path and each node-level computation computes the result and returns it towards the source node as shown in Figure 9(b). For the centralized algorithm, we assume that the sender does not know all the nodes along the path, but only knows its next-hop node towards the destination. The sender sends a query message to this node. The node replies with its local bandwidth information and next-hop node towards the destination. The sender then sends a query message to the next-hop node, and the node replies with its local information. The query-reply action continues until the sender sends the last query message to the destination, as in Figure 9(a).

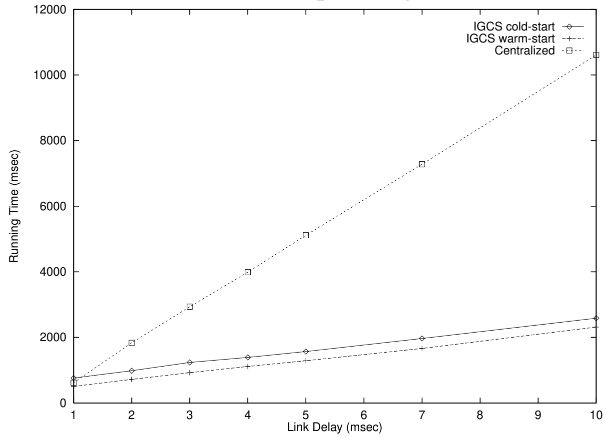
It is straightforward to show that the running time of the centralized algorithm grows as the square of the length of the path, while the running time of the IGCS algorithm is linear in the length of the path. The running time of both algorithms is linear with the path delay, though the coefficient is larger for the centralized algorithm.

The IGCS computation performs in-band signaling in the first iteration to initiate a node-level computation at each node. The initiation of a node-level computation includes creation of a thread and code loading for underlying and injected codes. The codes are loaded from a code server that has also been implemented in Bowman. The first cold-start computation experiences a delay incurred by the code loading. However, once the code modules are loaded and stored in the code cache at each node, the subsequent computations that utilize the code modules do only cache lookup, reducing the time for the computation initiation significantly. The cold-start overhead for the modules is about 35 msec, while the warm-start overhead is only about 5 msec.

Figure 10(a) shows the running time of each algorithm with different path lengths for a link delay of 5 msec. For the IGCS algorithm, we show both cold-start and warm-start results. The running time of the centralized algorithm is comparable to the IGCS results for a path length of two. The running time of the centralized algorithm, however, grows much faster than IGCS as the path length increases. At a path length of four, the centralized algorithm requires nearly twice as much time as IGCS; the gap increases quickly with more hops. The IGCS cold-start grows slightly faster than IGCS warm-start since, for the cold-start, the code loading time at each node is accumulated as the length of the path grows.



(a) Effect of path length



(b) Effect of link delay

Fig. 10. Running time with varying link delay and path length

Figure 10(b) shows the running time of each algorithm with different link delays. The length of the path is fixed at 10 hops. As discussed, the running time is a linear function of the link delay for both algorithms. The centralized algorithm, however, has a higher constant multiplier of $l(l+1)$, compared to $2l$ for the IGCS algorithm, where l is the number of hops. Although the IGCS cold-start and the IGCS warm-start show the same rate of growth, the cold-start has slightly more overhead due to its code loading time. Figure 10(b) confirms the measured difference in overhead for warm-start and cold-start.

The IGCS computation shows better scalability in terms of link delay and path length, compared to the centralized algorithm. Its running time is a linear function of link delay and path length with a small coefficient value. Although there is an overhead of code loading in the IGCS computation, it is negligible for any path of length larger than three hops.

V. Related Work

Prior work related to this paper can be separated into two broad categories: work in the virtual network community and work in distributed computing.

IGCS is a light-weight dynamic-configurable distributed system. However, the IGCS system is more suitable to a certain type of distributed algorithm due to its iterative gather-compute-scatter structure. There are at least two different styles of distributed algorithms that are naturally fit into the IGCS computation model: heartbeat algorithms and probe/echo algorithms. The IGCS system provides a distributed computing environment for those types of problems.

The heartbeat algorithm is suitable for the problems in the area of parallel iterative computations such as grid computation. As an example, consider the problem of computing the topology of a network. A distributed algorithm with heartbeat paradigm [5] works as follows: Each process executes a sequence of iterations. On each iteration, processes exchange their local knowledge of the topology with their neighbors and build their own local topology information using the exchanged information. As the iterations complete, each process gradually converges on the complete topology. The computation terminates when all the processes have learned the complete topology. This type of algorithm is called a heartbeat algorithm since the actions of each process are like the beating of a heat: first expand, sending information out; then contract, gathering new information in.

The probe/echo algorithm is suitable for distributed computations that explore over graphs. A probe is a query sent by one node to its successor; an echo is a subsequent reply to the query. For the same problem of topology discovery explained above, the probe/echo algorithm works as follows [5]: The initiator sends out the probe message to its all neighbors. After receiving a probe, a node sends it to all its neighbors except the node from which it receives the probe, and waits for an echo from each. To break the possible deadlock situation, if a node receives the multiple same probes, it immediately sends an echo containing a null topology. Eventually, a node will receive an echo in response to every probe it sends. At this point, it echoes the union of its neighbor set and the echoes it receives. In the probe/echo algorithm, only the initiator gets the complete topology information unlike the heartbeat algorithm. For computations that disseminate or gather information on graphs, probe/echo algorithms are more efficient that heartbeat algorithms [5]. In contrast, the heartbeat algorithms are appropriate and necessary for many parallel iterative algorithms in which nodes need to exchange information until they converge on an answer.

In the virtual network community, the NetScript active networking project at Columbia University addresses the deployment of virtual networks: in particular, as NetScript Virtual Networks (VANs) [6]. The NetScript effort focuses on a common router programming language, and their work describes the general notion of coordinated deployment and management. The VAN architecture abstracts underlying networked resources and connectivity into a single end-to-end programmable network to support uniform deployment, configuration management, security, and end-to-end resource management. The major goal of VANs, therefore, is to provide an abstraction of network resources that can be programmed and managed as a single object. The VANs architecture lacks the ability of providing customized virtual connectivity and interesting topology.

The X-Bone [7] is a generalized overlay management system. Its architecture consists of overlay manager, resource daemons, and a multicast control protocol. The X-Bone directory tool, `xd`, incorporates an interface and a resource discovery protocol, adapted from the `sd` and `sdr` M-Bone tools. The X-Bone also serves as a framework to coordinate and manage other new mechanisms (e.g. multi-protocol label switching) within overlay networks. A vision for the X-bone is that it could be used to automatically create virtual topologies that have certain (graph-theoretic) properties, e.g. map a eight node cycle to a certain physical topology such that the overall latency is minimized. In this paper, we have developed the IGCS computation model that generalizes the creation of such mappings.

Supranet [8] provides a toolkit that can be used at different levels, thus accommodating the needs of beginner, intermediate, and expert creators. The toolkit incorporates creating a topology, generating a routing table and defining security requirement over the topology. It focuses on optimizing the topology and resources of the virtual network to the physical network. Its optimization, however, is static in a sense that it occurs only at the creation time of the supranets and uses a fixed set of algorithms. For example, the toolkit automatically creates appropriate routing tables that reflect the creator's decision about the topology and the routing tables never change during the whole supranet lifetime. Our design is more general and dynamic. An IGCS computation is general enough to identify a wide variety of customized topologies and the indentified topologies can be dynamically managed by subsequent IGCS computations.

VI. Conclusions

Increasingly, situations arise in which it is desirable to query network state, identify network locations and instantiate virtual topologies. The black box interface to network topology that is currently provided by IP makes this difficult or impossible. Active networking, on the other hand, provides a programmable interface to networking resources, and thus the possibility of exposing some of the internals of the black box. We have used active networking to provide a programmable mechanism to identify topological entities and protocols to instantiate those entities into a virtual topology.

We have not addressed several key issues related to the querying of network state and the instantiation of multiple virtual topologies. Specifically, this paper does not consider resource management across virtual topologies (nor within a virtual topology). Some form of admission control for virtual topologies may be necessary, depending upon the resource guarantees that are desired. We have also neglected the various issues of security, including access control for node and link state and authorization to create virtual topologies. Resource management and security are both issues of importance for active networks (with or without virtual topologies), and we expect that solutions developed by the active networks community will be applicable to virtual networks.

References

[1] "Active Error Recovery (AER) : A reliable multicast implementation utilizing Active Network services," http://www.tascnets.com/panama/AER/.

[2] IETF IP Security Working Group, "IP Security Protocol," *Work in Progress*.

[3] A. Segall, "Distributed Network Protocols," *Trans. on Information Theory*, vol. 29, no. 1, pp. 23–35, Jan 1983.

[4] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert, "Bowman: A Node OS for Active Networks," *To appear in Infocom 2000*.

[5] Gregory R. Andrews, "Paradigm for Process Interaction in Distributed Programs," *ACM Computing Surveys*, vol. 23, no. 1, pp. 49–90, March 1991.

[6] Y. Yemini and S. da Silva, "Towards Programmable Networks," in *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, Oct. 1996.

[7] Joe Touch and Steve Hotz, "The X-BONE," in *Third Global Internet Mini-Conference in conjunction with Globecom '98*, Sydney, Australia, Nov. 8-12 1998.

[8] L. Delgrossi and D. Ferrari, "A Virtual Network Service for Integrated-Services Internetworks," in *7th International Workshop on Network and Operating System Support for Digital Audio and Video*, St. Louis (Missouri), May 1997.

APPENDIX: IGCS UNDERLYING AND INJECTED PROGRAMS

```
u_int i;
igcs_io_t *tmp_io;
igcs_msg_table_t *cur_in;

for (i = 0; i < igcs_get_iteration(); ++i) {

  /* Gather Phase */
  igcs_get_gather(i, tmp_io);   /* get channels for gather phase */
  if (tmp_io->num_ch != 0) {
    igcs_install_gather_filter(tmp_io); /* Install gather filter */
    cur_in = igcs_gather_msg(tmp_io);   /* get incoming messages */
    c_Ep(inMsg) = cur_in;               /* assign input messages */
    igcs_uninstall_gather_filter(tmp_io);/* Uninstall gather filter */
  }

  /* Compute Phase */
  igcs_raise_slot(Compute);         /* execute the compute slot */

  /* Scatter Phase */
  igcs_get_scatter(i, tmp_io); /* get channels for scatter phase */
  igcs_scatter_msg(c_Ep(outMsg),tmp_io);     /* scatter messages */
}
```

Fig. 11. IGCS Underlying Program

```
igcs_msg_t * tmp_msg;
igcs_io_t tmp_io;
int i;

memcpy(tmp_msg,                   /* get signaling message */
      (igcs_msg_t *)igcs_get_sigmsg(), tmp_sig->len);

i = igcs_next_hop(tmp_msg->src_id);/* set scatter phase of */
tmp_io.num_ch = 1;                 /* the second iteration */
tmp_io.ch[0] = i;
igcs_set_scatter(1, &tmp_io);

igcs_get_all_vn_channel(&tmp_io);  /* set scatter phase of */
_igcs_get_diff_channel(&tmp_io, i); /* the first iteration */
igcs_set_scatter(0, &tmp_io);
igcs_set_gather(1, &tmp_io);        /* set gather phase of 1st iter */

tmp_msg->src_id = net_utils_local_ip_number();
tmp_msg->type = IGCS_SIG;
c_Ip(outMsg) = tmp_msg;              /* set output message */
```

Fig. 12. IGCS Injected Program Example