RZ 3378 (# 93424) 15/10/01
Engineering & Technology 14 pages

Research Report

Design and Implementation of a Distributed-Agent-based Simulation for Hierarchical Service-Deployment

Abhishek Kumar*, Robert Haas†

*Indian Institute of Technology
Delhi
akumar@cse.iitd.ac.in
Work performed while the author was a summer student at IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

†IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
rha@zurich.ibm.com

Design and Implementation of a Distributed-Agent-based Simulation for Hierarchical Service-Deployment

Abhishek Kumar*, Robert Haas†

*Indian Institute of Technology, Delhi

†IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

Large and heterogeneous networks are difficult to manage, particularly when a wide set of customized services are expected to be deployed in such networks. This is especially true when parts of the network are programmable, such as Network Processors for instance. To help automate the deployment task of such services, a new protocol has been proposed. This paper describes the implementation aspects of the simulation used to evaluate the complexity of the protocol. The paper presents the design and usage of an agent-based extension to the network simulation software tool ns-2, as well as the implementation of the deployment of services of three categories, i.e., a distributed web-cache service, a service providing quality-of-service (QoS) using differentiated services (DiffServ), and a Virtual Private Network service providing QoS between multiple end-points, and also using DiffServ. A Straight Path Search algorithm is introduced, which reduces the amount of nodes queried, in order to perform optimal DiffServ deployment. In the VPN case, an adaptation of the Selective Closest Terminal First algorithm is presented that computes an approximation of the Minimum Steiner Tree over a hierarchical network.

1 Introduction

The rise of programmability levels within network elements has made the manual deployment of network services a far more complex task than it was in the past. Heterogeneous environments, in which networks elements offer widely differing capabilities, and large-scale networks composed of thousands of nodes require automated methods to deploy new services. In addition, competing Internet Service Providers (ISPs) require flexibility to quickly accommodate the needs of their customers, even for short time periods, with customized services provided by Internet Software Vendors (ISVs). Such requirements can hardly be fulfilled by the current centralized manual upgrades. Active networks promote the idea of user sessions defining the service required by their data packets in the network. The work described here focuses on network services such as web caching, QoS mechanisms, and Virtual Private Networks (VPNs).

The bottom line is to be able to make the best possible use of the underlying capabilities of the network elements, such as specialized processors lying on the data path (network processors), to deploy a service in the most favorable condition. Human intervention should be limited to the definition of high-level allocation policies, whereas the network itself should take care of the actual deployment.

This report is structured as follows. Simulation of the behavior of large-scale networks is more suitable than actual prototypes and testbeds; Thus a brief overview of existing simulation platforms, based on the literature and source-code evaluation, with their main drawbacks and/or advantages, is provided first.

Next, the design of the ns-2 simulation of the four steps of the deployment mechanism will be presented [HDS01]. Examples of the three types of services are introduced namely: path-based, node-based, and path-and-node-based service deployment. The algorithms used to execute the service deployment for each service are introduced.

2 Simulation

In this section, we describe the requirements for a suitable simulation platform, and then review the main existing simulation platforms and their respective features.

2.1 Simulation requirements

To allow verification and measurements of improvements, offered by the service-deployment mechanism, the simulation platform should be as flexible as possible, and offer good performance, namely,

- allow the simulation of large-scale networks (typically more than 1000 nodes);
- interpret topologies generated both manually and using more sophisticated tools (such as BRITE [MLMB01], GT-ITM);
- support physical and logical hierarchies;
- allow various path-selection algorithms at each level of the hierarchy;
- support uplinks for the creation of transition matrices;
- operate with time-related factors (various caching times);

- model transient behaviors (measure traffic to reroute/redeploy a service after a broken link or node);
- support the encoding of capabilities into nodes, varying over time when services are being deployed; and
- adapt the deployment of services based on actual clients' usage. For instance, a caching server could be relocated to minimize the average response time.

2.2 Analysis of simulation platforms

2.2.1 MARS (Maryland Routing Simulator) [ADZMS91]

- Supports static and dynamic routing.
- Automatic topology generation tools are not supported.
- The simulator is obsolete (1994-95).

2.2.2 TeD (Telecommunications Description language) [PNL96]

- New simulation paradigm, resembling VHDL.
- Supports simulations over large topologies.
- Detailed implementation of PNNI available.
- Simulation framework not as detailed as ns-2 or SSF.
- Automatic topology generation tools are not supported.
- Support for SMP-based parallel simulations available.

2.2.3 SSF: (Scalable Simulation Framework) [SSF01]

- A simulation environment optimized for parallel and distributed architectures.
- Three architecture levels of the simulator: The simulation framework, the protocol suite, and the network model to be simulated. The network to be simulated is described in domain modeling language (DML).
- Scalable to simulations of large networks.
- Will be supported by BRITE topology generation tool. No GT-ITM support, to our knowledge.
- Transient behaviour and time-related factors should be implementable.
- Capabilities of nodes can be extended, but related changes to DML will be necessary.
- No known support for logical hierarchies.

2.2.4 ns-2: Network Simulator 2 [ns-01]

- Simulation environment for serial execution.
- Large-scale simulations possible.
- Mixed architecture with C++ and OTCL implementation. The simulated model is described in TCL.
- Support for GT-ITM exists. BRITE support being developed.
- Transient and time-related behaviour can be implemented.
- No known support for logical hierarchies.
- Node capabilities can be extended by deriving from the 'Node' class.
- Application data transfer implemented for 'web-cache' simulations.

2.3 Simplifying assumptions

The process of creating groups of nodes in a hierarchical fashion is not modeled in the simulation (top-down, or bottom-up, with various parameters such as group size and number of levels). Changes in the topology (i.e., broken link, mobility of nodes) that might require an adaptation of the hierarchy after a certain divergence has been reached, are not modeled either.

3 Simulation Framework

Following the analysis of various simulation platforms, ns-2 was chosen to develop the simulations. ns-2 is written in C++, with an OTcl interpreter as a front-end. The simulation scripts, including the topology and the events, are in TCL. For our simulations, the C++ part of ns-2 has been extended to support Hierarchical Service Deployment (HSD) functionality in the form of *Agents* attachable to *nodes*.

3.1 Communication between agents

The HSD *agents* communicate using a customized, unreliable datagram service, which in turn uses the IP functionality of ns-2 to send packets. ns-2 does not support the inclusion of data in packets. To include specific information in packets being exchanged across the network, a new header type for corresponding agents has to be defined. In ns-2, packets are objects with many header fields and no data. But they are *simulated* to be carrying an explicitly specified amount of data and no header. It is a simple matter to calculate and specify the packet size to be simulated as the sum of header size and data size.

A new header type *hsd* is specified to include the HSD-specific data. This header carries the unique *deployment id*, the *service name* of the service being deployed, and the *iteration no* of the iteration to which the packet belongs. Apart from this, the header also carries a pointer to an object of class HsdData. Because the simulator runs in a single thread, it is possible to exchange data among the objects of the simulator by passing pointers to that data. All

deployment-specific data, e.g. the list of neighbor nodes between which the cost of a DiffServ-capable path is to be summarized, is stored in an object derived from the class HsdData and a pointer to this object sent in the packets. For overhead measurement, the simulated size of the packets is specified to be the sum of size of the object derived from the HsdData class and the sizes of the UDP and IP headers.

3.2 Underlying topology

The agents in the HSD framework represent a hierarchy of processes (daemons) running on physical nodes in the network. Correspondingly, in the ns-2 simulations the HSD Agents do not have an existence of their own, but are attached to nodes. The *nodes* and the *links* between these nodes are created in the usual way, well explained in the ns-2 documentation.

3.3 HSD agents

In the ns-2 simulations there are two types of HSD agents, physical and logical. Physical agents are present in the lowest level of the hierarchy and represent individual physical nodes in the hierarchy. Logical agents are present in all but the lowest level of the hierarchy, and represent a peer group of several nodes in level n at the next highest level, $n + 1$, of the hierarchy.

Usually all the nodes in the physical topology of the network have one physical HSD agent attached to them even though this is not necessary, non HSD-capable nodes can co-exist in the topology. The command to create a physical agent is:

`set p0 [new Agent/Hsd/Physical]`

and to attach it to a node is:

`$ns attach-agent $n0 $p0`

where `n0` is the node, `p0` is the physical agent being attached to it, and `ns` is the simulator instance.

Logical agents are created with a similar command:

`set l0 [new Agent/Hsd/Logical]`.

These agents are attached to a node by the same command:

`$ns attach` $< node >< agent >$.

In PNNI, the peer-group leader is elected by the members of a peer group, and represents the peer group at the next level in the hierarchy. Logical agents that act as peer-group leaders in the simulations are attached to one of the nodes in their underlying peer group. Note that this is not mandatory and it is perfectly valid to distribute logical agents randomly. The above restriction serves to maintain conceptual correspondence with PNNI hierarchies.

3.4 Adding hierarchical information

To obtain the relevant hierarchical information, each agent needs to know the identity of its parent (unless it is at the top of the hierarchy), its children (unless it is at the lowest level of the hierarchy) and its adjacent neighbors. The adjacent neighbors can belong to the same peer group or to a different one. For the latter, "suitable naming" as explained in the next subsection is crucial to maintain and propagate the correct hierarchical information across the network.

The peer-group leader to peer-group member relationship is equivalent to a parent-child relationship. This has to be explicitly mentioned by:

`$p0 parent $l0`

where agent `p0` is told to set `l0` as its parent, and:

`$l0 addchild $p0`,

where agent `l0` is told to add `p0` to its list of children. This has to be done for each parent-child pair at all levels of the hierarchy.

Furthermore, each physical agent is given the "suitable name" of all its adjacent neighbors. By suitable names we mean the following: Those adjacent neighbors that are part of the same peer group as the agent in question, are mentioned by their physical agent names. The adjacent neighbors from a different peer group are mentioned by the name of their first ancestor that shares a peer group with an ancestor of the agent in question. For example, in Fig. 1, physical agent A.2.1 is given the names of its neighbors as A.2.2 (link N2-N3), A.1 (link N2-N1) and B (link N2-N4). Similarly, agent A.1.1 is told that its neighbors are A.1.2 (link N0-N1) and B (link N0-N6).

The adjacency information is mentioned through the command:

`$`$< agent >$ `neighbor $`$< agent2 >$,

where $< agent2 >$ is the "suitable" name chosen, as explained above.

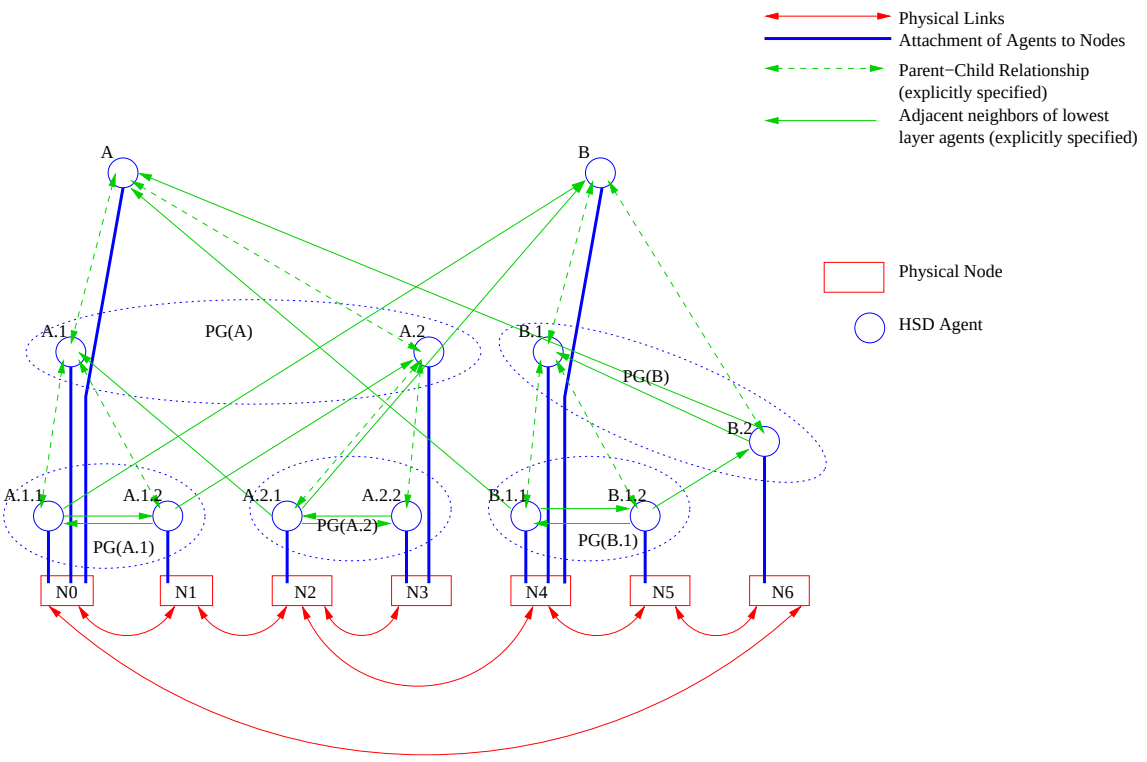

Figure 1: The logical hierarchy built on top of a physical topology.

3.5 Automatic generation of hierarchy information

The adjacency information is explicitly specified only for the physical agents. At the level of logical agents, this information is aggregated by deploying a special service called *hierarchy-build*. The solicit packets just trigger further solicitation down the hierarchy. The physical agents send the list of their adjacent neighbors in the summarize phase. The logical agents,

on receiving the summarized packets from all their children, identify all those neighbors of child agents that are not also child agents, as their own adjacent neighbors. This is the point where the naming scheme described in the previous section works out consistently. As an example, let us see what happens to logical agent A.2 in Fig. 2 in the summarized iteration. Child A.2.1 mentions A.2.2, A.1, and B as its neighbors, while child A.2.2 mentions only A.2.1 as its neighbor. Since A.2.1 and A.2.2 are already known to be children of agent A.2, it removes these from the merged list of neighbors received from its children and advertises the remaining agents, A.1 and B, as its adjacent neighbors. The logical agent A on receiving this information from A.2 will further remove A.1, known to be a child, and recognize B as an adjacent neighbor.

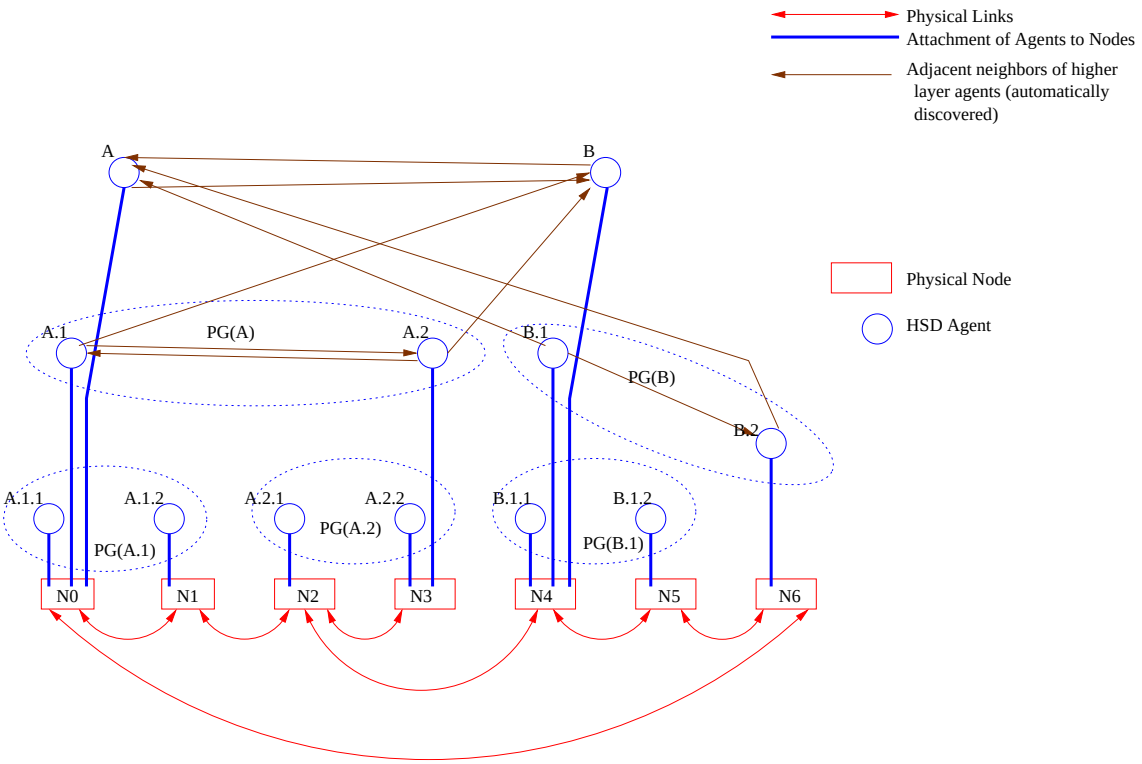

Figure 2: The automatically discovered adjacencies.

3.6 Further automation of topology and hierarchy generation

To run simulations for large topologies, the hierarchy generation will have to be further automated, and optionally linked with automatic generation of the topology. The two main classes of solutions considered are discussed below. The selection of the better solution and its implementation are left for future research.

3.6.1 Bottom-up hierarchy construction

This scheme envisages the use of a general-purpose topology generator, like GT-ITM or BRITE, to generate the underlying topology. This topology is the input to the hierarchy generator that runs a clustering algorithm to identify peer groups at successive levels of the hierarchy and names the nodes in the hierarchical naming format. Once this is done, the identification of "suitable" names of neighbors–as required by the *hierarchybuild* service–amounts

to identifying the common substrings of the hierarchical name strings in addition to the first point where they differ. The main problem here is to define and develop the right clustering algorithm, which is equivalent to hierarchical naming of all nodes in the topology.

3.6.2 Top-down hierarchy construction

This scheme envisages a combined automatic generator for both the hierarchy and the underlying physical topology. The input parameters might specify the number of hierarchical levels, the average number of nodes at each level (or equivalently, in each peer group), and the degree of connectivity of the graph (average node degree etc.). The generator should then produce an output in TCL (ns-2 format) that specifies the creation of all the nodes and links of the topology, the creation of physical and logical HSD agents and their attachment to the corresponding nodes, and lastly, the specification of the "correct" names of the neighbors for all physical agents. After deploying *hierarchybuild* all the agents should have the relevant hierarchical information.

A fractal-based idea of top-down hierarchy construction would be to have a basic graph repeated at each successive hierarchical level. The top peer group of the hierarchy looks exactly like the basic graph. Each logical node at this level represents a peer group at a lower level of the hierarchy, which in turn, looks like the same basic graph. The pseudo-regularity of such fractal-like graphs makes it easy to specify the uplinks with the "correct" names of neighbors. A preliminary implementation of this scheme was realized, and a topology size of up to 100 nodes, with a *basic* graph of 10 nodes, was generated and used in simulations.

4 Implementation of Three Different Service Types

In this section, we describe the design and usage of the extensions to the simulation tool to deploy three types of services. The deployment of a webcaching mechanism is considered first, as part of the Node-based deployment category. Then an example of DiffServ is introduced to illustrate the Path-based category. Finally, an example of multipoint DiffServ (providing QoS for a VPN) is presented, as an extension of the Path-based category.

4.1 Node-based service: webcache

Since webcache is not dependent on the horizontal links and uplinks through the hierarchy, this information need not be specified. Thus, after specifying the physical topology, we create and attach the HSD agents to the corresponding nodes and specify the parent-child information for each such pair. This completes the hierarchy specification for webcache.

4.1.1 Specifying attributes

Next, we specify the attributes of agents. Since only physical agents represent physical nodes in the network, attribute values for physical agents only, are specified. Logical agents can have summarized attributes for the peer group they lead, but these are not specified explicitly.

For webcache, we need to specify available CPU capacity at physical agents, which is done by the command:

$< *agent* > `cpucap` < *capacity* >,

where capacity is an integer value between 0 and 100.

4.1.2 Running a service deployment instance

This completes all the specifications required and we can trigger the deployment by the command:

`$`< *agent* > `deploy webcache .`

The < *agent* > here should be the root agent of the hierarchy.

4.2 Path-based service: DiffServ

DiffServ, as opposed to webcache, requires topological information (connectivity between nodes). Therefore, the format of summarized information sent by nodes to their parents peer-group leader is modified for the service DiffServ.

In this section, we describe how the topological information is adapted for the needs of the simulation, and then describe the algorithm used to execute the first step of the service-deployment procedure, namely $SelectAllNodesBetween()$ [HDS01]. The next step of the mechanism, namely summarization, is then described together with the data structures and algorithms required. The last two steps, dissemination and advertisement, are described as well.

4.2.1 Modified transition matrix with outer nodes indexing

The earlier format presented cost of an agent on level n in the hierarchy in terms of a transition matrix [HDS01]. In this transition matrix, $m_{i,j}$ represents the number of nodes on the shortest path between border nodes i and j (at level $n-1$ in the hierarchy) of the peer group. This information is sent by the parent agent (at level n) to the "grand-parent" agent (at level $n+1$). Thus the nodes at level $n+1$ receive summarized information from nodes at level n, which represent cost in terms of matrices, indexed by border nodes at level $n-1$. This means that the node at level $n+1$ has to be aware of the mapping between horizontal links at level n and level $n-1$, before it is able to compute its own transition matrix.

In order to facilitate such computations, transition matrices are modified so that they represent the cost of the shortest path between outer neighbor nodes at level n. Therefore, nodes at level $n+1$ only need know about the horizontal links at level n, and not about their mapping to level $n-1$.

For simplicity in implementation, the matrix is represented as a list, in which each entry mentions a pair of neighbors followed by the cost of linking those neighbors using a path passing through DiffServ enabled nodes only. A list representation of costs is ideal for the graph algorithms that need to be run to determine the shortest path for all pairs (Floyd-Warshall) and, later for multiple-endpoint DiffServ–the selective closest terminal first algorithm for computing the approximate minimum steiner tree.

4.2.2 Directed solicitation

In the first step of the mechanism, nodes are selected at each hierarchy level in order to direct the solicitation to the appropriate domains. It is for instance not necessary to solicit nodes belonging to a stub network if neither the source or destination, that needs to be connected with a DiffServ service, lie in this stub network. This function is called $SelectAllNodesBetween()$ in [HDS01].

Given a graph $\mathcal{G}(V, E)$ and a list of vertices, endpoints, this function finds all those vertices that lie on a *straight path* between any two vertices in the list of endpoints. A *straight path* between two endpoint vertices is defined as a set of vertices that forms a path between the two endpoints, and no subset of which forms a path between the same two endpoints. This function is implemented by modifying the Depth-First Search (DFS). This modified DFS returns one of three integer values: Determined negative (0), determined positive(2) or undetermined (1).

The algorithm Straight-Path Search (SPS) described next, sets the value *marked[u]* as TRUE if u lies on a straight path between *source* and *dest* and if not, as FALSE.

```
SPS(source,dest)
1   for each vertex u ∈ V[G]
2      do color[u] ← WHITE
3         marked[u] ← FALSE
4   SPS-Visit(source,dest)
```

```
SPS-Visit(u,dest)
1   localresult ← 0
2   color[u] ← GRAY
3   for each v ∈ Adj[u]
4      do if  v = dest
5         then localresult ← 2
6   if localresult < 2
7      for each v ∈ Adj[u]
8         do if color[v] = WHITE
9            then if NumGrayNeighbors(v) < 2
10              then localresult ← max(SPS-Visit(v,dest) localresult)
11              else localresult ← max(localresult,1)
12  if localresult = 2
13     then marked[u]=1
14  color[u] ← WHITE
15  return(localresult)
```

```
NumGrayNeighbors(u)
1   result = 0
2   for each v ∈ Adj[u]
3      do if color[v] = GRAY
4         then result++
5   return (result)
```

This algorithm is expected to be quite expensive because the same vertices are visited multiple times. To reduce the overhead, we modified the `SPS-Visit` function by caching the result at visited vertices, if a definite result is obtained there.

```
SPS-Visit-modified(u,dest)
1   localresult ← 0
```

```
2   color[u] ← GRAY
3   for each v ∈ Adj[u]
4      do if v = dest
5         then localresult ← 2
6   if localresult < 2
7      for each v ∈ Adj[u]
8         do if NumGrayNeighbors(v) < 2
9            then if color[v] = WHITE
10              then localresult← max(SPS-Visit-modified(v,dest) localresult)
11           else if color[v]= BLACK
12              then if marked[v]= TRUE
13                 then localresult ← 2
14           else localresult ← max(localresult,1)
15  if localresult = 2
16     then marked[u] ← 1
17  if localresult = 1
18     then color[u] ← WHITE
19     elsecolor[u] ← BLACK
20  return(localresult)
```

This modification is expected to decrease significantly the running time of the Straight-Path Search, although it wrongly marks some vertices as TRUE. This is the version currently implemented, with the expectation that the cost of soliciting the false positive vertices, will be made-up by the improvement in computational complexity.

4.2.3 The summarization step

The summarized information to be sent back to the soliciting agent is the cost matrix, as described at the beginning of this section. Since the problem is to compute the cost of the shortest path between multiple source-destination pairs, we use the Floyd-Warshall all-pairs shortest path computation algorithm. This algorithm computes a $(n \times n)$ matrix $\mathcal{D}$ in $O(n^3)$. An added advantage of choosing this algorithm is that a predecessor, matrix Π, can be calculated in $O(n^3)$ time. This matrix can then be used to generate a list of vertices through which the shortest path between any pair of vertices passes.

In the actual implementation there is an added complexity due to the fact that the metric cost for DiffServ is not the sum of edge weights but the cost of the nodes on a path. For complex nodes, at higher levels in the hierarchy, the cost of including the node in the path depends on the neighboring nodes. This can be translated to the standard problem with edge weights as the cost metric, by representing each edge in the original graph as a vertex and each vertex in the original graph as a set of edges connecting each of the vertices in this new graph that represent edges incident in the original graph. For example, the edges between A and B.2, and B.2 and B.3 in the original graph in Fig. 3 are represented by the vertices A,B.2 and B.2,B.3, respectively in the transformed graph in Fig. 4. The cost of going from A to B.3 through B.2 in the original graph is equivalent to the cost of the edge from A,B.2 to B.2,B.3 in the transformed graph. For each endpoint vertex in the original graph, an extra vertex is added in the transformed graph, with links of no cost to any of the vertices in the transformed graph that represent edges incident on this endpoint vertex in the original graph. Note that

paths can only originate from or terminate at these extra vertices. Traversal of these extra vertices is not allowed.

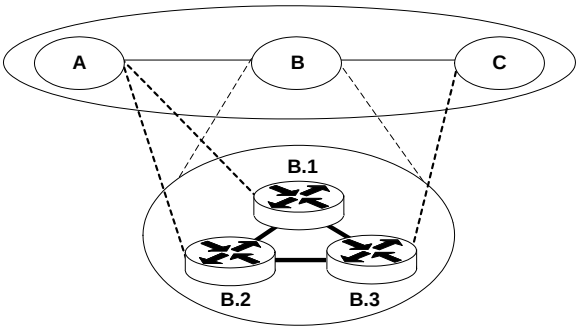

Figure 3: A topology example of the Floyd-Warshall algorithm.

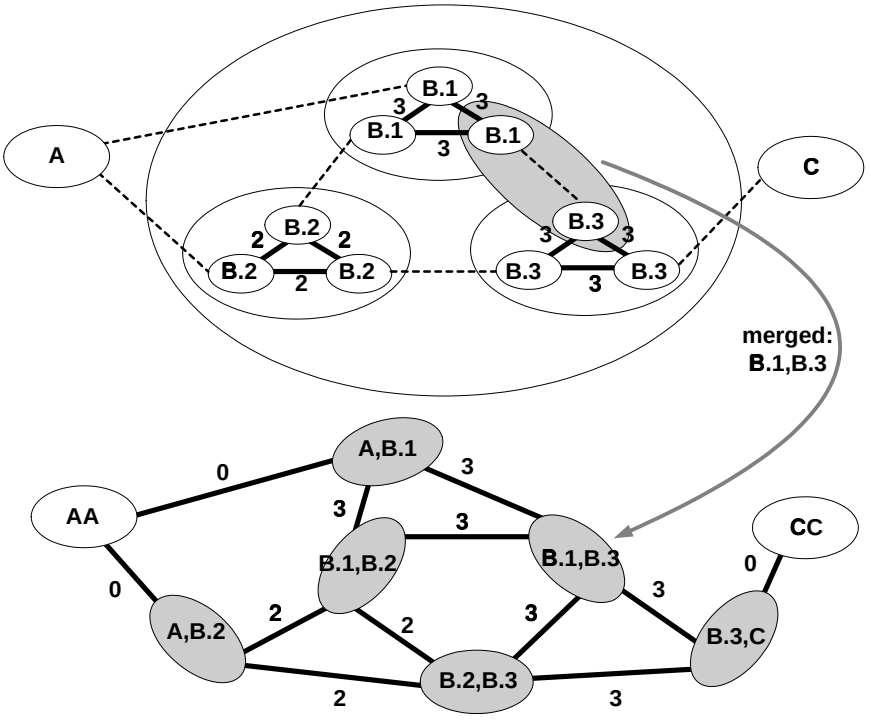

Figure 4: The extended original graph and its corresponding transformed graph.

The construction of a path from the predecessor matrix Π also changes. We add an intermediate matrix that is the predecessor matrix for the new (translated) graph. The predecessor matrix is still retained, but now the element $\Pi_{i,j}$ in this matrix is the vertex in the original graph, that on which edges i and j, which are vertices in the new graph, were incident. This is illustrated in Fig. 3. The distance matrix $\mathcal{D}$ is:

	AA	A,B.1	A,B.2	B.1,B.2	B.2,B.3	B.1,B.3	B.3,C	CC
AA	-	0	0	2	2	3	5	5
A,B.1	0	-	5	3	5	3	6	6
A,B.2	0	5	-	2	2	5	5	5
B.1,B.2	2	3	2	-	2	3	5	5
B.2,B.3	2	5	2	2	-	3	3	3
B.1,B.3	3	3	5	3	3	-	3	3
B.3,C	5	6	5	5	3	3	-	0
CC	5	6	5	5	3	3	0	-

The first row of the intermediate matrix $\mathcal{I}$ is:

	AA	A,B.1	A,B.2	B.1,B.2	B.2,B.3	B.1,B.3	B.3,C	CC
AA	-	AA	AA	A,B.2	A,B.2	A,B.1	B.2,B.3	B.3,C

Entry $\mathcal{I}_{i,j}$ is the vertex before vertex j, in the transformed graph, on the path from vertex i to vertex j.

Similarly, the first row of the predecessor matrix Π is:

	AA	A,B.1	A,B.2	B.1,B.2	B.2,B.3	B.1,B.3	B.3,C	CC
AA	-	A	A	B.2	B.2	B.1	B.3	C

Entry $\Pi_{i,j}$ in the predecessor matrix corresponds to the vertex in the original graph that lies between the vertices j and $\mathcal{I}_{i,j}$ in the transformed graph. In other words, both j and $\mathcal{I}_{i,j}$ represent edges in the original graph, and $\Pi_{i,j}$ is the vertex in the original graph on which both of them are incident.

4.2.4 The dissemination step

At the end of the summarize iteration, the root agent in the hierarchy already knows the cost of the shortest path between the source destination pair in its $\mathcal{D}$ matrix and the path in the $\diamond$ matrix as well as that of the intermediate matrix $\mathcal{I}$. The root agent now selects the child agents on this path and sends them a disseminated packet containing their neighbors which were also on this path. The child agent, if it is a logical one, repeats the process sending disseminated packets to those children that lie on the shortest DiffServ-capable path between the neighbors mentioned in the disseminate packet. Lastly, the physical agents, on receiving disseminated packets, verify if the neighbors mentioned in the disseminated packets are indeed adjacent neighbors and if they are DiffServ-capable. If yes, they enter the advertisement step.

4.2.5 The advertisement step

The advertisement functionality implemented, sends back a packet to the parent agent if the disseminated service was successfully installed. For DiffServ, the physical agents send back an advertise packet to their parent agents. The logical agents, on receiving advertise packets from each of their children, send an advertise packet of their own to their parent agents. The root agent prints a message of successful deployment and ends the service deployment instance.

4.3 Mutliple path-based: Multipoint DiffServ

The third service for which hierarchical deployment is simulated, is a multiple endpoint differentiated service. This is a multiple-destinations version of the service discussed in the previous section and substantially reuses the implemented code for *DiffServ*.

4.3.1 Finding the optimum tree

The problem of finding the optimum tree cost that connects a specified subset of vertices in a graph, is the Minimum Steiner Tree computation problem, known to be NP complete for the case where cost is an additive edge weight. We use an approximation algorithm, with polynomial time complexity, that finds a solution guaranteed to be, at worst, twice as expensive as the optimum, known as the Selective Closest Terminal First (SCTF) algorithm [Ram96] with $k = |V|$. Since the all-pairs shortest-path computation is already done in the summarize phase, the time complexity is $\mathcal{O}(mlog(mn))$ where m is the number of endpoints and n the number of vertices in the graph.

4.3.2 The dissemination step

After receiving the summarized metric costs from the solicited children during the solicitation step, the root agent computes the minimum tree cost, connecting the client endpoints using the algorithm described above.

Note that the solicited information is not complete for the minimum steiner tree (MST) computation. The summarized information from the child nodes only gives the cost of connecting any pair of its neighbors through itself. The gains of branching within a child node are not known, and hence sub optimal decisions might be made. We expect any such gains to be of the order of magnitude of the cost of traversing the node and as such not to significantly degrade the cost of the resulting tree. Other lossy aggregation mechanisms such as star representation or star representations with extra links could be candidates to be explored in the future.

The logical agents successively repeat this process for their own peer groups in the dissemination step. The physical agents, on receiving the disseminated packets, behave in the same way as they do for DiffServ and initiate the advertise iteration, which again is a replication of the advertise iteration of DiffServ.

5 Conclusion

The design and parts of the implementation used for the simulation of automated service-deployment have been described. As an artifact of the mechanism, the generation of the necessary abstracted information of the topology was implemented as any other service. The resulting information is used by all path-based services that need to be aware of connectivity in order to perform a proper deployment.

To direct solicitation through the hierarchy towards useful nodes, an algorithm called Straight-Path Search is introduced. It allows the solicitation of nodes contained in stub networks or loops to be avoided. A succinct discussion of the complexity of the algorithm is presented.

An adaptation of the Selective Closest Terminal First algorithm, used to find an approximation of the Minimum Steiner Tree is introduced. Its use over a hierarchical network is discussed.

The three services implemented, namely webache, DiffServ, and multipoint DiffServ, make use of these algorithms in order to achieve deployment with a minimum number of solicitation messages, and an acceptable complexity to optimality ratio.

References

[ADZMS91] Cengiz Alacttinoglu, Klaudia Dussa-Zieger, Ibrahim Matta, and A. Udaya Shankar. Mars(maryland routing simulator) - version 1.0 user's manual, June 1991. University of Maryland.

[HDS01] R. Haas, P. Droz, and B. Stiller. Distributed service deployment over programmable networks. In *DSOM'01*, Nancy, France, October 2001.

[MLMB01] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Bayers. Brite: Universal topology generation from a user's perspective, April 2001. Computer Science Department, Boston University.

[ns-01] Network simulator ns2. http://www.isi.edu/nsnam/ns/, September 2001.

[PNL96] Brian J. Premore, David M. Nicol, and Xiaowen Liu. A critique of the telecommunications description language (teD). Technical Report TR96-299, Dartmouth College, 1996.

[Ram96] S. Ramanathan. Multicast tree generation in networks with asymmetric links. *IEEE/ACM Transactions on Networking*, 1996.

[SSF01] SSF, scalable simulation framework. http://www.ssfnet.org/homePage.html, September 2001.