ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Computer Networks 41 (2003) 269–284

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Configuring sessions in programmable networks

Sumi Choi [a,*], Jonathan Turner [a], Tilman Wolf [b,1]

[a] Department of Computer Science and Engineering, Campus Box 1045, Washington University, St. Louis, MO 63130-4899, USA
[b] Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003, USA

Received 26 February 2002; received in revised form 1 October 2002; accepted 8 October 2002

Responsible Editor: A. Campbell

Abstract

The provision of advanced computational services within networks is rapidly becoming both feasible and economical. We present a general approach to the problem of configuring application sessions that require intermediate processing by showing how the session configuration problem can be transformed to a conventional shortest path problem for unicast sessions or to a conventional Steiner tree problem for multicast sessions. We study both a capacity-constrained version of the problem and an unconstrained version and show, through a series of examples, that the method can be applied to a wide variety of different situations. Particularly, we show how to extend Dijkstra's shortest path algorithm for use on the constrained version, and show that this approach can make significantly better use of network resources.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Routing; Programmable networks; Session configuration

1. Introduction

Advances in technology are making it possible to incorporate general purpose processing capabilities in network routers. Network processor components with more than 10 RISC cores have become available and are starting to appear in high performance routers from several different equipment vendors. Research in *active networking* [2,6,8] is exploring the potential of programmable routers, and other approaches are being pursued by individual router vendors.

This paper is concerned with the problem of how to map application sessions onto network resources, when those network resources may include computational elements that perform some service on behalf of the applications. For example, a video application might invoke a video compression service in the network to reduce its use of network bandwidth. There may be several places in the network where the required compression and decompression service could be performed. We would like to select the best locations that meet the application's requirements. In this paper, we describe a general methodology for configuring

* Corresponding author.
E-mail addresses: sycl@arl.wustl.edu (S. Choi), jst@arl.wustl.edu (J. Turner), wolf@ecs.umass.edu (T. Wolf).
[1] This work was conducted while the author was at Washington University in St. Louis.

1389-1286/03/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S1389-1286(02)00396-1

such applications so as to make most effective use of network resources, including link bandwidth and the computational resources provided by the network. Our methodology is not restricted to systems in which application services are provided at routers. It can also be used to configure application services provided by network-attached servers.

We assume an operating environment in which application sessions are explicitly configured when the application starts up. The configuration of an application session includes selection of intermediate processing nodes and the network links used for communication among the various components of the application. In our view, this session-oriented approach is needed to enable efficient allocation of network resources among competing applications. This is especially true for applications that require a certain level of resources in order to achieve an acceptable quality of service. However, even "best-effort" applications can benefit from a resource allocation system that seeks to configure applications to take advantage of locations where resources are plentiful, rather than simply letting them compete for resources in locations where the required resources may be scarce.

Sections 2–6 describe various application scenarios that each raise different resource configuration issues. In each case, we show how the problem can be reformulated so that it can be solved in a similar fashion with the approach that we propose. In Sections 7 and 8, we discuss how to handle sessions that require explicit resource reservation. We provide an overview of related work on resource allocation and configuration in Section 9 and conclude in Section 10.

2. Routing through one processing site

We start with the simplest version of the application configuration problem. In this version, we have two participating end systems and there is some intermediate computation that is to be performed somewhere in the network (possibly a format translation, for example, allowing two otherwise incompatible end systems to share information). There are a number of sites within the network where the processing could occur, but not all of the sites may be able to perform the needed processing (perhaps they are not capable of executing the required program, or perhaps their computational resources are already fully committed to other tasks). The application configuration system must select one of the sites within the network and select network paths joining the end systems to the intermediate processing site. It should do this in such a way as to minimize the use of network resources, including link bandwidth and processing "bandwidth".

We can state the problem formally as follows. The network is represented by a directed graph, $G = (V, E)$, in which the nodes correspond to routers and end systems, while the edges correspond to links. Let $R \subseteq V$ be a subset of the nodes that represent sites where intermediate processing may occur. For brevity, we will refer to these as *red* nodes. Each edge (u, v) has an associated *cost* $c(u, v)$ and each red node r has an associated cost $c(r)$. Finally, we have a source vertex s and a destination vertex t. Our objective is to find a least-cost path from s to t that includes at least one red node. The cost of a path is the sum of the costs of its links, plus the cost of the least cost red node along the path. Note that the overall path from s to t may not be a simple path. See Fig. 1 for an example of the problem. The *red* nodes can be distinguished from the other nodes by the numbers that indicate their processing costs. The heavy weight edges in the figure indicate the best path from s to t that passes through at least one red node.

There is one fairly obvious approach to solving the problem. First, solve the single-source shortest

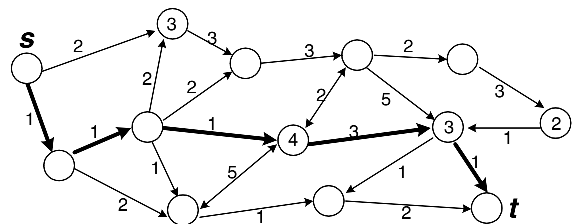


Fig. 1. Network with processing sites.

path problem [5] from s to all other nodes, considering link costs only. Second, solve the single-destination shortest path problem to t from all the other nodes. At the end of these two steps, for each vertex u, we know the cost of the shortest path from s to u and from u to t. So we can simply iterate over all nodes $r \in R$ and select the node that minimizes

$$d(s, r) + d(r, t) + c(r),$$

where $d(x, y)$ denotes the length of the shortest path between x and y, considering just the edge costs. For a graph with n vertices and m edges, this algorithm can be implemented to run in $O(m + n \log n)$ time. This is the same complexity as for finding a shortest path in a graph, so we cannot expect to improve on it substantially. The only real drawback of this method is that it does not readily generalize to more complex situations. For that reason we consider an alternative approach that can be applied more generally.

Our alternative approach is to transform the original problem to a conventional shortest path problem on a different graph. We then solve this new problem using standard methods and apply the results back to the original problem. The first step in the transformation is to make two copies of the original graph G. We refer to these two copies as *layers* in the resulting graph and identify them as layer 1 and layer 2. For each vertex u in the original graph, let u_1 denote the copy of u in layer 1 of the target graph and let u_2 denote the copy of u in layer 2. The edges in the two layers have the same costs as the corresponding edges in the original graph. Now, for every node $r \in R$, we add an edge (r_1, r_2) in the target graph and let $c(r_1, r_2)$ be equal to the cost originally assigned to r. This completes the construction of the target graph. See Fig. 2 for an illustration of the construction. To solve our original problem, we simply find a shortest path from s_1 to t_2 in the target graph, considering link costs only (see Fig. 2). The resulting path can then be mapped back to a path in the original graph by "projecting" the two layer path onto a single layer.

The correctness of this procedure is easily established. First, note that the least cost path from s_1 to t_2 does correspond to a path (not necessarily a simple path) in the original graph and the cost of the path is the same as the cost defined in the original problem statement for the corresponding path in the original graph. Second, note that there cannot be a cheaper solution to the original problem. If there were, this solution would have to correspond to a path from s_1 to t_2 in the target graph that is cheaper than the given least-cost solution, a clear contradiction.

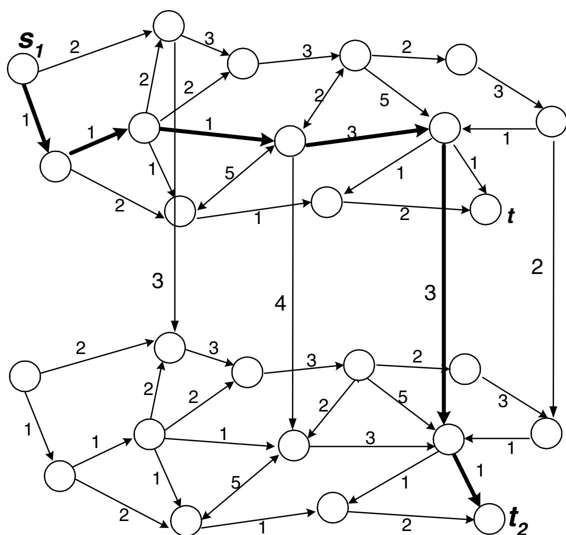


Fig. 2. Transformed network for single site processing.

3. Routing through multiple sites

We now consider a more general application configuration problem. There are again two participating end systems, but here there are several intermediate computational steps that are to be performed at possibly different locations in the network. For each step, there may be multiple sites where the processing could be done. One simple example is secure data transmission, where the intermediate processing steps include encryption and decryption processing. The encryption processing can be done at any of several nodes in the originating end system's domain and decryption processing can be done at any of several nodes in the destination end system's domain. We allow k intermediate processing steps for any $k \geqslant 1$.

We can state the problem formally as follows. The network is represented by a directed graph, $G = (V, E)$, with each edge (u, v) having an associated cost $c(u, v)$. As before, we have a source node s and a destination node t. For $1 \leqslant i \leqslant k$, let $R_i \subseteq V$ be a subset of the nodes. R_i contains sites where the ith intermediate processing step may be performed. Accordingly, each node $r \in R_i$ has an associated cost $c_i(r)$. We define an *admissible path* from s to t to be a path (not necessarily simple) that includes nodes from each of the R_i, appearing in order. That is, a path $u_1, u_2, \ldots, u_m$ is admissible, if there are integers $i_1, \ldots, i_k$ that satisfy $1 \leqslant i_1 \leqslant \cdots \leqslant i_k \leqslant m$ and $u_{i_j} \in R_j$ for $1 \leqslant j \leqslant k$. The list of nodes $(u_{i_1}, \ldots, u_{i_k})$ is called a *site list* for the path. An admissible path may have multiple site lists. Note that a node may appear in a site list more than once. The cost of a site list is the sum of the costs of its nodes and the cost of an admissible path is the sum of the costs of its edges, plus the cost of its least expensive site list. Fig. 3 shows an example of the problem. In this figure, nodes drawn with "thick" circles are in R_2, while the other nodes containing numbers are in R_1.

A brute force approach to solving this problem involves enumerating all possible combinations of processing nodes and connecting them with the shortest paths. However, the number of possible combinations grows proportionally to n^k, making this approach impractical, even for modest values of k.

Fortunately, the problem can be solved efficiently by reducing it to an ordinary shortest path problem in a different graph. The target graph G has $k + 1$ layers, each layer being just a copy of the original graph, and numbered from 1 to $k + 1$. For each node u in the original graph, we let u_i denote the copy of u in layer i. Now, for every node $r \in R_i$, we add an edge (r_i, r_{i+1}) in the target graph and let $c(r_i, r_{i+1})$ be equal to the cost $c_i(r)$ assigned to r in the original graph. See Fig. 4 for an example of a target graph for a problem with $k = 2$. To solve the original problem, we find the shortest path from s_1 to t_{k+1} in the target graph. The resulting path can be mapped back to a path on the original graph by "projecting" the path back onto the original graph. In the projected path, the site list consists of nodes each of which corresponds to an inter-layer edge used in the shortest path.

The correctness of the procedure can be shown in a similar fashion as in Section 2. Consider the least cost path from s_1 to t_{k+1}. It is easy to see that it corresponds to an admissible path in the original graph and that its cost is the same as the cost of the admissible path. Also note that there can exist no cheaper solution to the original problem. Any cheaper solution would have to correspond to a path from s_1 to t_{k+1} in the target graph, yielding a contradiction to the definition of the shortest path.

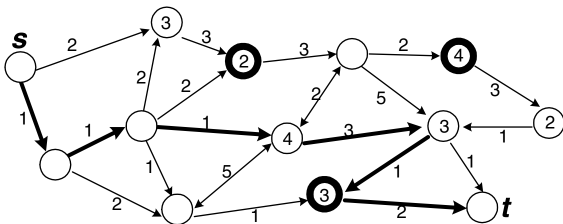


Fig. 3. Network for multiple site processing.

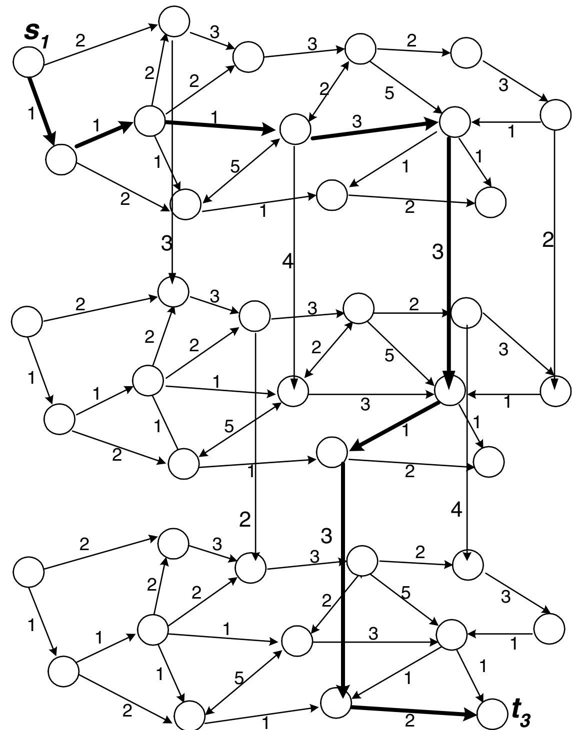


Fig. 4. Transformed network for multiple site processing.

4. Applications that alter bandwidth

Certain processing steps performed on behalf of an application may alter properties of the data. For example, processing steps that compress data can change its bandwidth requirements by substantial amounts. We would like to be able to configure compression and decompression processing in the network, so as to best exploit the savings that can be obtained, while simultaneously accounting for the costs associated with the compression algorithm itself. More generally, we want to be able to configure arbitrary applications that modify the bandwidth requirements of the processed data stream. Examples for applications that decrease the bandwidth of a stream are data and image compression, filtering, and data merging. Applications that increase the bandwidth of a data stream are data and image decompression, forward error correction coding, certain encryption and authentication schemes, etc.

To quantify the changes in bandwidth, we define the *bandwidth scale factor* γ_i for processing step i to be the ratio of the outgoing bandwidth to the incoming bandwidth for processing step i. The application configuration problem introduced in the previous section can be generalized to handle changes in bandwidth requirements. With the cost of links linear to the data bandwidth, the only change needed is to the definition of the cost of an admissible path, to account for the changes in the bandwidth of the data stream. Let $P = u_1, \ldots, u_m$ be an admissible path, that includes the site list $L = (u_{i_1}, \ldots, u_{i_k})$. The cost of P with respect to site list L is given by

$$\sum_{j=1}^{i_1-1} c(u_j, u_{j+1}) + c(u_{i_1}) + \sum_{j=i_1}^{i_2-1} \gamma_1 (c(u_j, u_{j+1}) + c(u_{i_2})) + \sum_{j=i_2}^{i_3-1} \gamma_1 \gamma_2 (c(u_j, u_{j+1}) + c(u_{i_3})) + \cdots + \sum_{j=i_{k-1}}^{i_k-1} (\gamma_1 \gamma_2 \cdots \gamma_{k-1})(c(u_j, u_{j+1}) + c(u_{i_k})) + \sum_{j=i_k}^{m-1} (\gamma_1 \gamma_2 \cdots \gamma_k) c(u_j, u_{j+1}).$$

The cost of a path P, is the minimum over all site lists L of P, of the cost of P with respect to L.

The solution method of the previous section can also be generalized to handle bandwidth scaling. The target graph is constructed as before, but the edge costs of the target graph are modified as follows. For edges within layer i, the edge costs are multiplied by $\gamma_1 \gamma_2 \cdots \gamma_{i-1}$. Edge costs from layer i to layer $i+1$ are multiplied by $\gamma_1 \gamma_2 \cdots \gamma_{i-1}$. We solve the problem, as before, by finding a shortest path from s_1 to t_{k+1}.

5. Optional processing

Some network applications provide services that are not necessary for correct data transmission, but which can improve the performance or quality of the connection. These optional processing steps might decrease the transmission cost to some destination nodes, but not necessarily to all. We now extend our method to handle such cases.

For concreteness, we use a simple example of a compression/decompression application. The processing for compression and decompression incurs a cost, but the intermediate data stream has a lower bandwidth ($\gamma < 1$) which yields lower transmission costs. Thus, for long-distance transmissions the processing overhead is worthwhile, while for short distances, the cost of the added processing may exceed the benefit. The problem can be solved using the method of the previous section. To make the compression and decompression processing optional, for each vertex u in the original graph, we add edges (u_1, u_3), linking layers 1 and 3. These edges are assigned a cost of zero. Note, that for this method to work correctly, the bandwidth of the decompressed data stream must match that of the original, uncompressed data stream. In this case, we can actually use a slightly simpler target graph with just two layers, and edges (u_1, u_2) for all vertices $u \in R_1$ and edges (v_2, v_1) for all vertices $v \in R_2$. The edges within layer 2 are scaled by the compression factor, as are the edges from layer 2 to layer 1. In this case, the shortest path from s_1 to t_1 yields the best

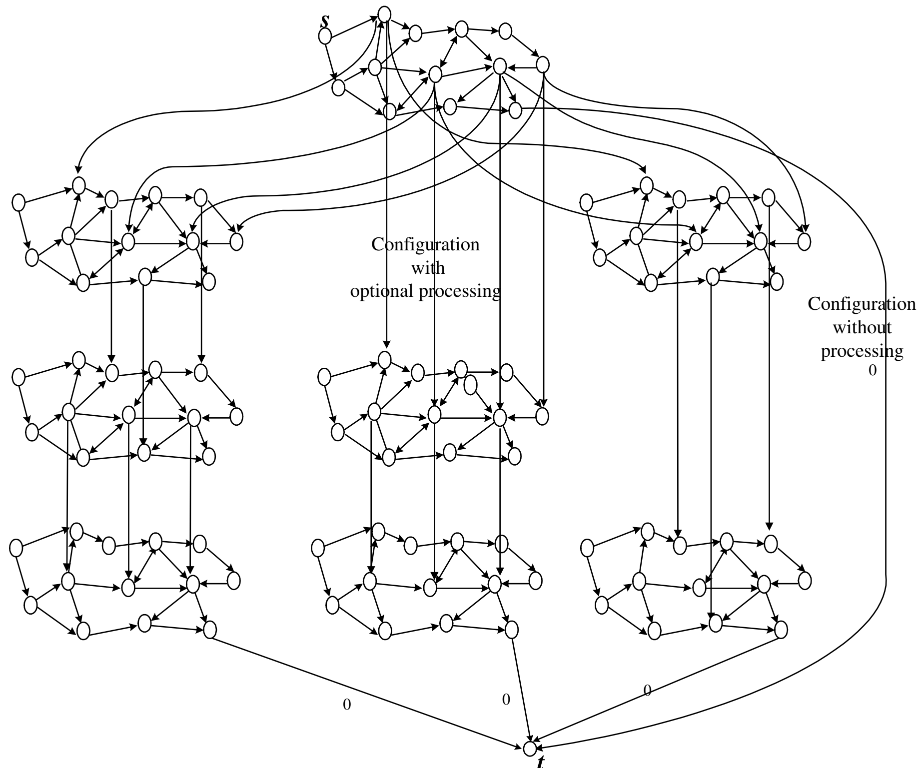


Fig. 5. Transformed network for optional processing.

configuration. If the path passes through layer 2 then compression is performed at the nodes corresponding to the selected inter-layer edges.

The method can be extended to configuring sessions where different processing stages are optional. However, when the effects on the bandwidth of the data stream are more complex than in the simple compression/decompression example, a more complex target graph may be required. These more general cases can be solved using target graphs that have a source node s connected to multiple columns of layers, where each column contains some subset of the layers for the complete processing, and eventually connected to the destination t below the last layer of each column. The general form of such a graph is illustrated in Fig. 5. The columns of layers connected from the source s and to the destination t represent possible choices of processing sequences.

6. Configuring multicast sessions

So far, we have considered several types of different application configuration problems with two participating end system and the common objective to find an optimal path from one to the other. In this section, we show that our method can be applied to multicast applications where there are multiple destinations, rather than just one. For each of the source–destination paths, we want to include the same sort of processing that we might apply to a unicast application. Our objective is to find a way of selecting processing sites and links so that the processing requirements are met, and so that the overall cost is minimized.

We illustrate the application of the method to multicast situations by considering a video distribution application, where a source sends stored compressed video on which decompression processing is performed at intermediate nodes, and

then the decompressed video is delivered to multiple receivers. As discussed earlier, we can solve this problem for unicast applications using a two layer graph with "decompression edges" from layer 1 to layer 2. The same target graph can be used for the multicast problem, where we have a source and multiple destinations. See Fig. 6 for an example of the target graph. The only real difference is that the objective of the problem becomes finding a least-cost subtree of the two layer network with the source at the root, and the destinations at the leaves. This problem is a Steiner Tree problem (as is the usual multicast routing problem), which is known to be NP-complete [9]. More precisely, our case belongs to the directed Steiner Tree problem where the graph contains directed edges, and this particular subproblem is also known to be NP-complete. While there is no known practical approximation algorithms for the directed Steiner Tree problem, some methods currently used for the conventional multicast tree problem may easily be applied to our multicast configuration problem. We do not discuss such methods further here; we simply note that they can be applied to finding an appropriate tree in the target graph, and we can then use this to produce a solution to the original multicast session configuration problem.

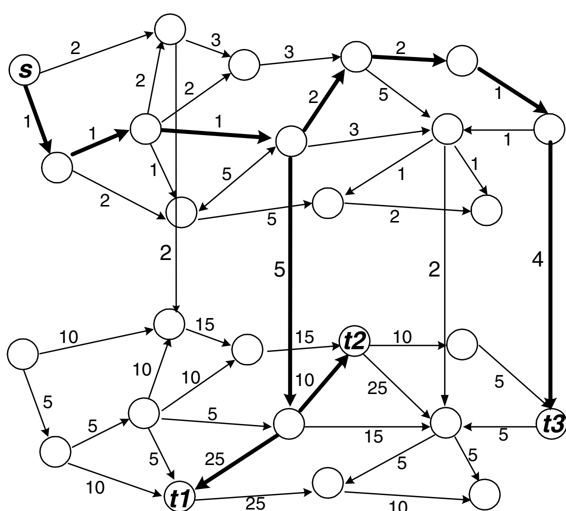


Fig. 6. Transformed network for multicast with compression.

7. Configuring sessions in capacity constrained networks

So far, we have not explicitly raised the issue of resource reservation and capacity constraints. One way to account for the finite nature of network resources is to simply omit from consideration those links and nodes that lack sufficient capacity to handle a given application session. So for example, if an application session requires 100 Mb/s of link bandwidth, we can simply remove from the network graph all edges corresponding to links with less than 100 Mb/s of available bandwidth. Similarly, if a session requires the equivalent of 50 MIPS of "CPU bandwidth" at one processing site, we can omit all nodes that have less than 50 MIPS of available capacity. Unfortunately, this does not quite work, since the problem specifically allows edges and nodes to be used more than once by an application session. If our example session were to use a given link twice, that link would require 200 Mb/s of unused bandwidth in order to accommodate both uses. Similarly, a node that is used in more than one processing step, must have sufficient available CPU bandwidth to handle all the steps for which it is used.

Unfortunately, there appears to be no efficient solution to this problem that can always find the best solution. Consider an instance of the session configuration problem in which every node but the source and sink is a potential processing site. There are $n-2$ intermediate processing steps (where n is the number of nodes in the graph, including the source and sink), but each node has only enough CPU bandwidth to perform one step. Problem instances like this correspond directly to the $\mathcal{NP}$-hard Hamiltonian path problem, so we cannot expect to find an efficient algorithm that is even guaranteed to find a feasible solution, let alone a least cost solution.

While we cannot solve the capacity-constrained version of the problem in all cases, we can incorporate capacity constraints into our path search, so that in most situations we can find low cost session configurations with the required resources. In this section, we show how this can be done, and demonstrate using simulation, that valid session configurations are rarely missed.

We focus here on unicast sessions that specify a processing requirement for each step and a bandwidth requirement for each path segment between two consecutive steps. As described in Section 4, the costs in the different layers are scaled to account for such effects.

We use the same layered graph model discussed earlier. However, to find shortest paths in the layered graph, we use an extension of Dijkstra's shortest path algorithm, that we call Dijkstra's algorithm with "capacity tracking". The algorithm ensures that no path in the shortest path tree constructed by the algorithm exceeds the available capacity of any resource. We start with a brief review of Dijkstra's shortest path algorithm.

Given a graph, and a source node s, Dijkstra's algorithm computes a *shortest path tree* rooted at s. Initially, the tree contains just s. The algorithm maintains a set S, of *boundary vertices*, which includes all nodes v that are connected to a vertex u in the partial tree constructed so far, by a directed edge (u, v). At the start of the algorithm, S contains the nodes v, for which there is an edge of the form (s, v). The algorithm also maintains, for each vertex v, a *tentative distance* $d(v)$, which is the length of the shortest path from s to v that has been found so far. It also maintains a *tentative parent* $p(v)$, which is the predecessor of v in a path from s of length $d(v)$. The quantities $d(v)$ and $p(v)$ are not defined for nodes that are neither in the tree, nor in S.

At each step, Dijkstra's algorithm selects a node v in S for which $d(v)$ is minimum, and adds it to the tree. It then examines each edge (v, w). For each node w that is neither in the tree nor in S, it adds w to S, setting $p(w)$ to v and $d(w)$ to $d(v)$ plus the length of (v, w). For each node w that is in S, it compares $d(w)$ to $d(v, w)$ plus the length of the edge (v, w), and if it finds that $d(w)$ is larger, it updates $d(w)$ and $p(w)$. If the set of boundary vertices is represented using a Fibonacci heap [5], Dijkstra's algorithm runs in $O(m + n \log n)$ time, where n is the number of nodes in the graph, and m is the number of edges.

When Dijkstra's algorithm is applied to a layered graph, some of the paths in the shortest path tree may contain edges on different layers that correspond to the same link or router in the original network, from which the layered network was constructed. This may lead to over-use of resources. To prevent this, we modify the basic processing step, to include a check for over-used resources. In particular, when a node v_i is added to the tree (i denotes the layer in which the vertex appears), we consider edges of the form (v_i, w_i) and (v_i, v_{i+1}). Before processing an edge of the form (v_i, w_i), we examine the path in the tree from s to v_i and add up the capacities required by all edges on the path that correspond to the original link (v, w). If this total capacity, plus the capacity that would be used by the edge (v_i, w_i) exceeds the available capacity of the link, then no action is taken with respect to that edge. Edges of the form (v_i, v_{i+1}) are handled similarly. We refer to this procedure as *capacity tracking*.

The extra time required by *capacity tracking* is $O((km)(kn))$, in the worst-case, where m and n are the number of edges and nodes in the original network and k is the number of processing steps. This can be seen by noting that the checking procedure is invoked no more than $k(m + n)$ times and each execution requires that we traverse a path with no more than $kn - 1$ edges.

The running time can be improved by maintaining an additional variable $\kappa(v_i)$ for each vertex in the partial tree constructed so far. If $u_i = p(v_i)$, then $\kappa(v_i)$ is the sum of the capacities required from all edges on the tree path from s to v_i that are copies of the link (u, v) in the original network graph. Similarly, if $v_{i-1} = p(v_i)$, then $\kappa(v_i)$ is the sum of the capacities required from all edges on the tree path from s to v_i that correspond to the server v in the original network graph. Using these additional variables, we can terminate the capacity tracking search from a node v_i back to s early, reducing the total time taken for *capacity tracking* to $O(kmn)$.

In practice, the extra time required by *capacity tracking* is much smaller than the worst-case analysis suggests, because networks are designed to have small diameter, which means that the paths in the shortest path tree generally have far fewer than kn edges. If we let D denote the maximum number of edges in a path from the root to a vertex in the shortest path tree, then the extra time

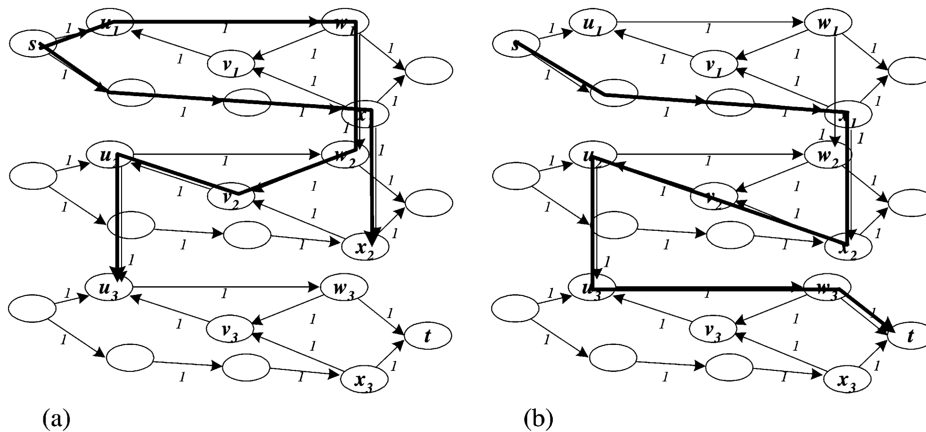


Fig. 7. Blocked path in capacity tracking. (a) Blocked path by capacity tracking. (b) A valid configuration not chosen by capacity tracking.

required by *capacity tracking* is $O(kmD)$. Even this result over-states the time required by *capacity tracking* in practice. As will be seen later, running time measurements show that *capacity tracking* takes less than double the time required by the original layered graph algorithm in more realistic situations.

Capacity tracking ensures that paths found by the algorithm do not over-use any resources. However, since the problem is $\mathcal{NP}$-hard, we cannot expect it to always find a valid path, even when a path exists. Consider the example shown in Fig. 7(a). If each link in the original network graph has one unit of capacity and the session requires one unit of capacity on each edge of the selected path, it can fail to find a path, as shown in part (a) of the figure. The bold edges are the edges that form the shortest path tree, at the time the path search terminates. Note that there is no way to extend the tree further, since the only edge leaving vertex u_3 has already been used in the top layer, and hence cannot be used again. On the other hand, there is a path that could be used for this session, as shown in part (b).

8. Simulation results

We performed a set of simulations for the session configuration problem to evaluate *capacity tracking*, the heuristic algorithm presented in the previous section. We also evaluated the following algorithms for performance comparison.

- *Strict resource accounting*: This algorithm simply implements the layered graph algorithm on the network graph from which the links and processing nodes that lack sufficient capacity to handle the maximum possible use by a given session are omitted. So for example if a session has two processing steps that require a total of 200 MIPS of processing capacity, then we omit inter-layer edges corresponding to servers with less than 200 MIPS of available capacity.
- *Loose resource accounting*: This is not a practical algorithm but provides a bound on the performance of realistic algorithms. It includes edges in the layered graph if they have sufficient resources to be used even once by the application. This may result in selecting paths for sessions that overuse resources.
- *Static shortest path*: This method uses the shortest path between a pair of nodes, without considering competing traffic. If any resource on the static shortest path lacks sufficient capacity to handle the session, this method fails. Because of this fixed routing, configuration attempts are likely to be failed when resources are heavily used.

Simulations were performed using four different network topologies.

- *Torus*: This network is based on a grid of 64 nodes where every node has an outgoing edge to each of its four neighbors, north, south, east and west along the grid lines. The nodes at edges of the square grid also have links that "wrap around" to the corresponding node at the opposite edge, resulting in a torus topology. The network has 128 edges and each is assigned an equal cost and capacity. Fig. 8 shows the network topology. A random subset of the nodes are designated as servers, with the ability to perform processing. All servers have the same capacity and are shown as triangles in Fig. 8. Note that there can be multiple shortest paths between any two nodes in *Torus*. The *static shortest path* routing randomly fixes one of the shortest paths for routing sessions between each pair of nodes.
- *Random*: This network is a random regular network with 64 nodes, each having four incident edges. We build the network starting with a random degree-bounded tree that spans all 64 nodes, then we expand the network by adding edges randomly until every node has exactly four incident edges. Again, every link has the same capacity and the same cost. A random subset of the nodes are designated as servers, with the ability to perform processing. All servers have the same capacity.
- *Metro 20*: is a more realistic network configuration, spanning the 20 largest metropolitan areas in the United States. The network topology is shown in Fig. 9(a). Nodes that are capable of performing processing are shown as triangles. Link costs are set equal to the physical distance between the nodes they connect, reflecting the higher cost associated with links spanning greater distances. The link capacities are selected to be large enough to handle the anticipated traffic.

  The link dimensioning procedure used for this purpose is taken from [7], which describes a constraint-based network design methodology and an interactive network design tool that implements it. We constrain the traffic in two ways. First, the total traffic entering and leaving a node is chosen to be proportional to the population of the metropolitan area represented by that node. Next, for each node u, we constrain its traffic to every other node using constraints that are proportional to the populations of the metropolitan areas represented by the other nodes. Specifically, if δ_v is the fraction of the population outside node u, that is associated with node v, then we limit the traffic between u and v to be no more than $1.3\delta_v$ times the total traffic entering and leaving node u. The factor of 1.3, was chosen to allow for some flexibility in the distribution of traffic, reflecting the natural variations that occur in network traffic. Given these traffic assumptions and a *default path* joining each pair of vertices, link dimensions can be computed using linear programming. The resulting link capacities guarantee that any traffic pattern satisfying the traffic constraints can be carried if the traffic is routed along the default paths. The default path between a pair of vertices is a shortest path containing at least one server, and can be found using a two layer network. The servers along each default path are dimensioned to handle the worst-case traffic load allowed by the traffic constraints. When performing the simulations, we do not constrain the traffic to use just the default paths, but the link dimensions are chosen, under the assumption that the default paths are used. However, the *static shortest path* method directly restricts sessions to use the default paths.
- *Metro 50*: is a larger version of the *Metro 20* network. It has a node for each of the 50 largest metropolitan areas in US. The topology is shown in Fig. 9(b). The links and servers are dimensioned in the same way as *Metro 20*.

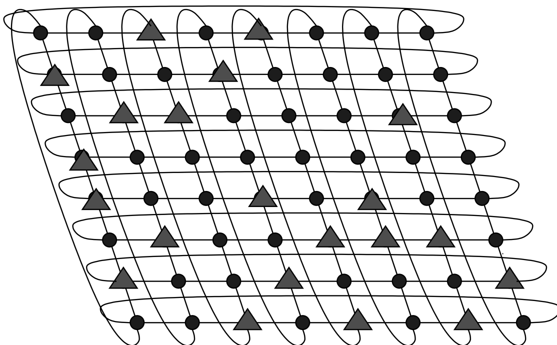
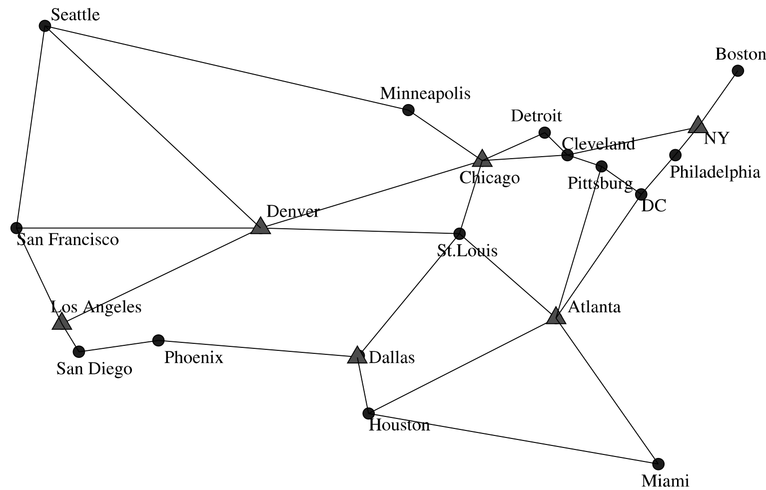
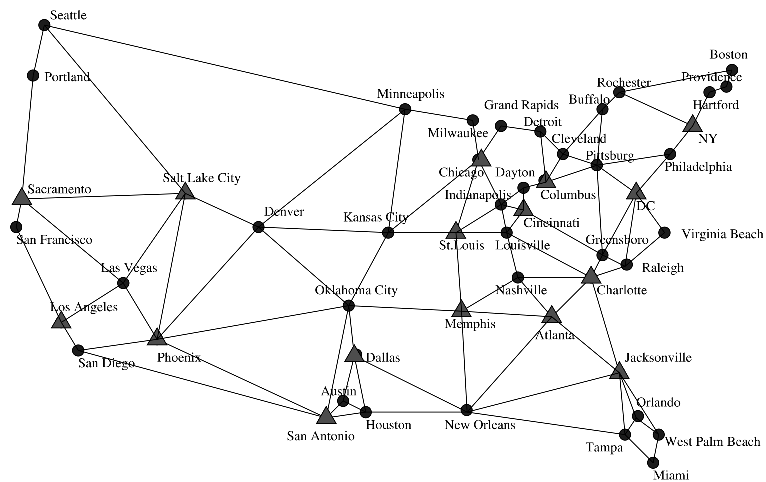


Fig. 8. Torus network.



(a) Metro 20 Network



(b) Metro 50 Network

Fig. 9. Metropolitan area networks. (a) Metro 20 network. (b) Metro 50 network.

While *Torus* and *Random* are not particularly realistic network configurations, they provide a more ‘‘neutral’’ context for evaluating the session configuration algorithms than the somewhat idiosyncratic network topologies that arise from real world considerations. By considering a variety of different networks, we hope to avoid drawing conclusions that may be attributable purely to special properties of a particular network.

There are several configuration parameters that affect the simulation results:

- *Density of servers* (P): The density of servers is just the ratio of the number of nodes that can perform processing to the total number. In the results reported here $P = 1/3$. The servers were randomly selected for *Torus* and *Random* and were configured for *Metro 20* and *Metro 50* as

shown in Fig. 9(a) and (b), where servers are drawn as triangles.

- *Session capacity requirement* (BW_s): The capacity that an individual session uses at each link and server; BW_s, is set to 3% of the average link capacity.
- *Number of steps* (N_{steps}): The number of processing steps that a session requires. In most of the results reported here, we set $N_{steps} = 3$.
- *Offered load at links* (O_l): The average offered background traffic level on each link. The simulation is done by generating background traffic levels independently at each link and node, then attempting to connect random pairs of nodes. This procedure was repeated multiple times to produce the reported results. Each simulation run included over 2.5 million session setup attempts. The background traffic was generated using an $M/M/k/0$ queueing model (k servers and zero length queues, where k is the ratio of link capacity to session bandwidth).
- *Offered load at servers* (O_p): The average offered background traffic level at each server. For the results reported here, the offered load at the servers is the same as the offered load on the links.

The selection of the end nodes of the sessions, was done completely randomly for *Random*. For *Torus*, the selected node pairs were restricted to be exactly four hops apart. For *Metro 20* and *Metro 50*, the selection of the end nodes was weighted by the populations of the cities, reflecting the higher traffic volumes expected in larger cities.

Our primary performance metric is the blocking probability, which is the percentage of session configuration attempts that were unsuccessful. Fig. 10 shows the blocking probabilities for the various algorithms, as a function of the offered load. The plots also show high blocking probability when the *static shortest path* method is used.

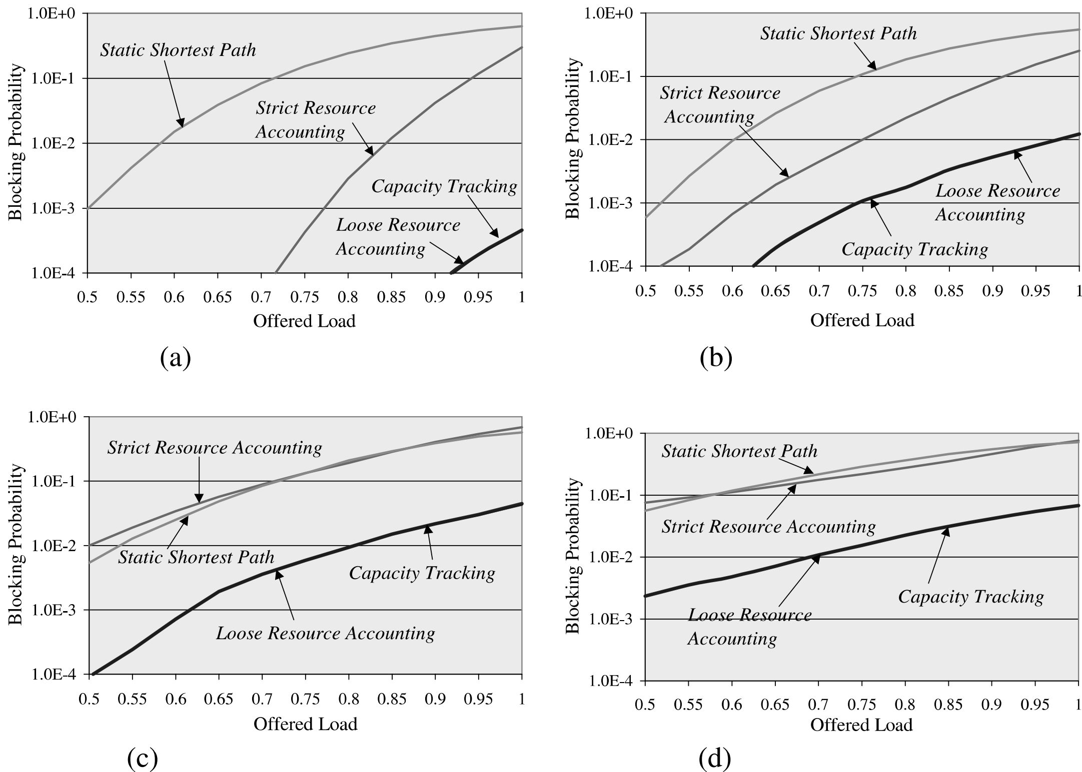


Fig. 10. Heuristics for session configurations. (a) Torus network. (b) Random network. (c) Twenty metro areas. (d) Fifty metro areas.

Recall that the default paths were used in the dimensioning process, so this restriction is worth considering, as a point of comparison. In general, however, the lack of routing flexibility implied by this policy results in higher blocking probabilities than with the other algorithms.

For all four networks, *capacity tracking* shows significant improvement over *strict resource accounting* and performs almost as well as *loose resource accounting* which is included as an idealized bound on algorithm performance.

For the *Torus* network, *capacity tracking* results in blocking probability less than 1% for load up to 80% (see Fig. 10(a)). Note that *Torus* has many paths between selected end nodes, and therefore, there is a good chance for algorithms to find valid paths while avoiding saturated links. Even then, *strict resource accounting* does relatively poorly, apparently because it often unnecessarily omits from the layered graph resources (primarily servers) that may be required in some feasible paths. Note that when all feasible paths are blocked by *strict resource accounting* it fails to find any configuration.

For *Random*, all algorithms experience higher blocking probability than for *Torus*. The explanation appears to be the variety of paths available between endpoint pairs in *Torus* and the limited separation between endpoints in the *Torus* simulation. In the *Random* network, endpoints were simply selected at random, so many pairs are likely to be further apart than the four hops that constrained the choice of endpoint pairs in the *Torus* simulation. In *Random*, there also tends to be fewer good "second-choice" paths, when the preferred path is not available.

For the more realistic *Metro 20* and *Metro 50* networks, blocking probabilities are generally higher. For *Metro 20*, we note that many sessions must take "detours" to pass through servers. For example, consider sessions between Pittsburgh and DC or Seattle and Minneapolis. When the default path is too busy to accommodate sessions, the "second-choice" paths typically require even longer detours. With *Torus*, on the other hand, the second and third choices are often no worse than the default. For *Metro 50*, the detours required to reach servers are generally smaller, but the number of hops required between endpoints tends to be larger; for example, there are 11 hops in the shortest path from New York to Los Angeles. Note that with *capacity tracking* blocking probabilities of less than 1% are obtained for offered loads of more than 70%.

We also measured the cost of the successful configurations. In Fig. 11, we show the configuration cost from all algorithms relative to the cost of the default shortest path, which is a lower bound. All algorithms provide nearly optimal costs at low loads, but deviate significantly at higher loads. For the *Metro 50* network, the paths produced using *capacity tracking* generally stay within about 10% of the lower bound up to loads of 95%. For the *Metro 20* network the cost rises to about 20% above the lower bound at a load of 95%.

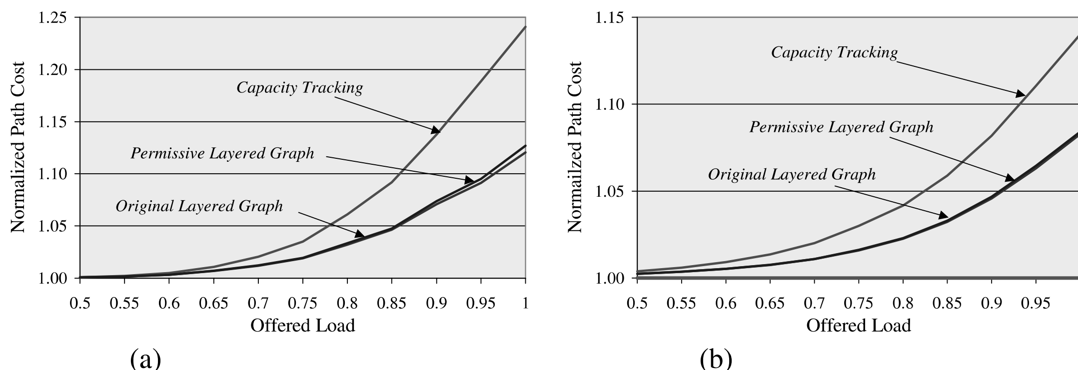


Fig. 11. Configuration cost. (a) Twenty metro areas. (b) Fifty metro areas.

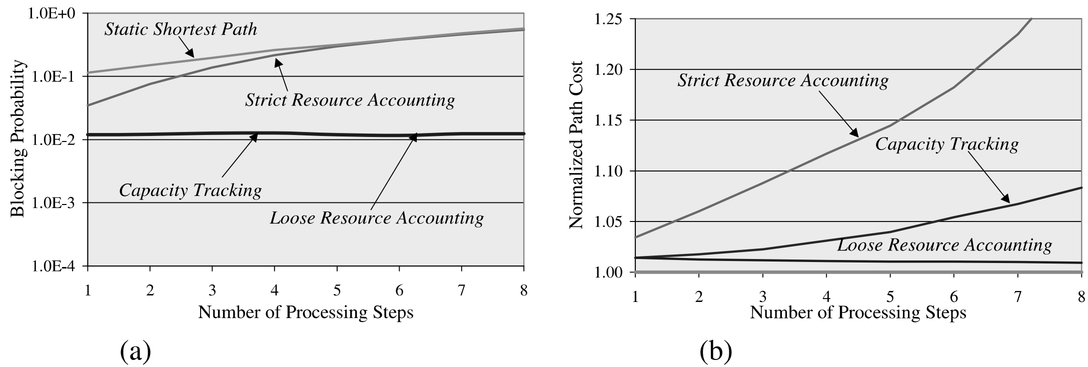


Fig. 12. Configurations at 75% traffic load for 50 metro areas. (a) Blocking probabilities. (b) Configuration cost.

In another set of simulations, we varied the number of processing steps while fixing the offered load at 75%. Fig. 12(a) shows the effect of this on blocking probability for *Metro 50*. As we increase the number of processing steps, *strict resource accounting* has more sessions blocked due to unnecessary elimination of resources from consideration, while *capacity tracking* experiences no increase in blocking. Fig. 12(b) shows the effect of increasing the number of processing steps on the path quality.

Lastly, we measured the average time required for session configuration by the different algorithms. Fig. 13 shows the results for *Metro 50*. For all algorithms, we varied the number of steps from 1 to 10. We observe that *capacity tracking* has a computational cost less than twice that of *strict resource accounting* algorithm when ten processing steps are performed. Considering that sessions are likely to have far fewer than 10 steps in the vast majority of applications, the superior blocking probability achieved with *capacity tracking* more than compensates for the extra computational time.

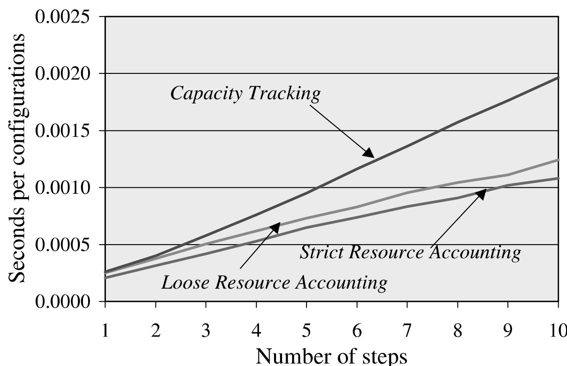


Fig. 13. Time requirements for session configuration.

9. Related work

Resource discovery and resource allocation are important elements of network programmability. The Darwin project [4] proposes a set of resource management mechanisms that support customized network services. Their resource management system is divided into four components, high-level resource allocation, run-time resource management, hierarchical resource scheduling and low-level resource allocation. Within their system, a *service broker* component called Xena provides both resource discovery and allocation. Xena formulates the resource allocation problem as a general optimization problem with multiple metrics. While this provides a very flexible and general formulation, it makes it computationally infeasible to find optimal solutions, even in simple situations. By contrast, our approach sacrifices some degree of flexibility to enable rapid computation of optimal solutions in the most common cases.

Different approaches are taken in [1,3]. Chae et al. [3] proposed a method for network discovery where they focused on identifying topological properties related to services and resource states.

Constrained network programmability is then provided to applications based on these properties. In their work, topological properties are determined by distributing network queries and then aggregating results back at the source, using a form of network fusion operation.

Another approach using market-based resource control mechanisms is considered in [1]. In this work, resources are treated as trade goods, network nodes and links as producers and applications as consumers. Service brokers are used to mediate access to resources between producers and consumers, using a form of currency exchange, and enable varying levels of competition and co-operation.

10. Summary

The provision of advanced computational services within networks is rapidly becoming both feasible and economical. Such services, either by routers or by network-attached processing sites, are potentially a significant benefit for network users, as they can relieve individuals from the need to acquire, install, and maintain software in end systems to perform required services. As such network services become more widely used, it will become increasingly important for service providers to have effective methods for configuring applications sessions so that they use resources efficiently.

We have presented a general approach to the problem of configuring application sessions that require intermediate processing. The method involves transformation of the original problem to a conventional shortest path problem. We have shown, through a series of examples, that the method can be applied to a wide variety of different situations. We have also addressed an issue raised by sessions that reserve resources and proposed an algorithm followed by the simulation study that showed positive prospects of the algorithm. To make the ideas in this paper directly applicable, it will be necessary to automate the methodology, so that resource management software can automatically determine the best way to configure a session to satisfy its requirements. The next step in reaching this objective is to develop a general way of specifying application requirements for intermediate processing, that is expressive enough to describe typical application scenarios, while being simple enough for application programmers to use effectively.

We believe that given such a specification method, it will be possible for network resource management software to combine information about network resource availability and an application specification, to produce a graph that represents the possible configurations of the application. By solving the appropriate optimization problem on this graph (typically a shortest path problem), the network resource management software will be able to automatically map the application to an appropriate set of resources. This paper represents a crucial first step in a research program that aims to achieve this objective.

References

[1] K.G. Anagnostakis, M.W. Hicks, S. Ioannidis, A.D. Keromytis, J.M. Smith, Scalable resource control in active networks. in: The Second International Working Conference on Active Networks, October 2000, pp. 343–357.

[2] A.T. Campbell, H.G. De Meer, M.E. Kounavis, K. Miki, J.B. Vicente, D. Villela, A survey of programmable networks, Computer Communication Review 29 (2) (1999) 7–23.

[3] Y. Chae, S. Merugu, E. Zegura, S. Bhattacharjee, Exposing the network: Support for topology sensitive applications, in: Proceedings of IEEE OpenArch 2000, March 2000, pp. 65–74.

[4] P. Chandra, A. Fisher, C. Kosak, T. Ng, P. Steenkiste, E. Takahashi, H. Zhang, Darwin: customizable resource management for value-added network services, in: Sixth International Conference on Network Protocols, October 1998, pp. 177–188.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, McGraw-Hill, New York, 1990.

[6] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, B. Plattner, A scalable, high performance active network node, IEEE Network 13 (1999) 8–19.

[7] H. Ma, I. Singh, J. Turner, Constraint based design of atm networks, an experimental study, Washington University Computer Science Department Technical Report WUCS-97-15, 1997.

[8] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, G.J. Minden, A survey of active network research, IEEE Communications Magazine 35 (1) (1997) 80–86.

[9] P. Winter, The Steiner problem in networks: A survey, Networks 17 (1987) 129–167.

Sumi Choi received the B.S. degree in Mathematics from Yonsei University in 1994 and M.S. degree in Computer Science from Brown University. She is currently a Ph.D. candidate in Computer Science and Engineering at Washington University in St. Louis. Her research interests include programmable networks, routing algorithms, resource management and network design.

Jonathan S. Turner received the M.S. and Ph.D. degrees in Computer Science from Northwestern University in 1979 and 1981. He holds the Henry Edwin Sever Chair of Engineering at Washington University, and is Director of the Applied Research Laboratory. The Applied Research Laboratory is currently engaged in a variety of projects ranging from Dynamically Extensible, Networks to Optical Burst Switching.

He served as Chief Scientist for Growth Networks, a startup company that developed scalable switching components for Internet routers and ATM switches, before being acquired by by Cisco Systems in early 2000.

Professor Turner's primary research interest is the design and analysis of switching systems, with special interest in systems supporting multicast communication. His research interests also include the study of algorithms and computational complexity, with particular interest in the probable performance of heuristic algorithms for NP-complete problems.

Turner is a fellow of ACM and a fellow of the IEEE. He received the Koji Kobayashi Computers and Communications Award from the IEEE in 1994 and the IEEE Millenium Medal in 2000. He has been awarded more than 20 patents for his work on switching systems and has many widely cited publications.

Tilman Wolf received his Diploma in Informatics from the University at Stuttgart, Germany in 1998. From Washington University in St. Louis, he received a M.S. in Computer Science in 1998, a M.S. in Computer Engineering in 2000, and a D.Sc. in Computer Science in 2002. He is currently Assistant Professor in the Department of Electrical and Computer Engineering at the University of Massachusetts at Amherst. His research interests are advanced computer networks, programmable routers, network processor design, and benchmarking.