Configuring Sessions in Programmable Networks with Capacity Constraints

Sumi Y. Choi and Jonathan Turner
Department of Computer Science, Campus Box 1045
Washington University, St. Louis, MO 63130-4899
{syc1, jst}@arl.wustl.edu

Abstract— **The provision of advanced computational services within networks is rapidly becoming both feasible and economical. As computational services become popular, it is important to have effective methods for configuring application sessions so that they use resources efficiently.**

In this paper, we discuss the problem of configuring application sessions that require intermediate processing. The problem was introduced in an earlier paper, where we showed how to optimally configure sessions in programmable networks by reducing the session configuration problem to the problem of finding a shortest path in a special graph constructed for the particular problem. This *layered graph method* **is quite flexible and can handle a variety of specific session configuration problems. However, it does not explicitly model limits on link bandwidth or processing capacity. In this paper, we show that the optimal session configuration problem is $\mathcal{NP}$-hard when capacity is constrained. Nevertheless, we have found efficient heuristics for which the network performance closely approximates the performance that can be achieved with optimal session configurations.**

I. Introduction

In this paper, we discuss the problem of configuring resources to support application sessions in programmable networks. Specifically, we study how to most efficiently allocate link bandwidth and the processing capacity of programmable network elements. This problem was introduced in an earlier paper [1], where we showed how to efficiently find the least-cost configuration for unicast sessions, under the assumption that there are no hard limits on the capacity of the various resources. In this paper, we extend the earlier work to incorporate capacity constraints that arise naturally when resources must be reserved for use by individual application sessions.

The motivation for programmable networks comes from two forces. First, continuing advances in technology are making it possible for network elements to be constructed using programmable components, even in high performance systems. Programmable components are attractive, because they make systems more flexible and make it easier to correct deficiencies that may be discovered only late in a product development cycle. This growing use of programmable components in high performance routers also creates an opportunity to perform more complex, application-specific processing of packets as they move through a network.

The second force motivating programmable networks, is the desire of network operators to provide more advanced services, and in general to make communication easier to use and hence more attractive to an increasingly broad and technically unsophisticated user population. The ability to embed application-specific functionality within network elements has the potential to enable a wide range of new applications which may not be technically feasible or economically viable otherwise.

There is a variety of possible approaches to realizing the potential that programmable network technology has to offer. *Active networking* [2], [3] is one such approach. In the best-known variant of active networking, the so-called *capsule model*, packets are interpreted by routers as programs to be executed, rather than just data to be forwarded. More realistic variants of active networking involve the dynamic execution of trusted programs on behalf of individual application sessions, in response to signaling messages exchanged at the start of a session [4]. This work is oriented toward the latter view of programmable networks. However, it can also be applied in networks where the use of programmable elements is controlled by administratively-determined policies, rather than user-initiated signaling messages.

To illustrate the potential of programmable networks, consider an example application. It involves an enterprise network with multiple sites that communicate over the Internet. As a matter of corporate policy, it is required that all communication between corporate sites be encrypted. This policy can be implemented by configuring encryption modules in programmable network elements, to encrypt all packets that go between two sites. This might be combined with modules for compressing video going between sites, to conserve the use of Internet bandwidth.

In [1], we showed how to optimally configure sessions in programmable networks by reducing the session configuration problem to the problem of finding a shortest path in a special graph constructed for the particular problem. This *layered graph method* is quite flexible and can handle a variety of specific session configuration problems. However, it does not explicitly model limits on link bandwidth or processing capacity, which is a significant limitation. In this paper, we show that the optimal session configuration problem is $\mathcal{NP}$-hard when capacity is constrained. Nevertheless, we have found efficient heuristics for which the network performance closely approximates the performance that can be achieved with optimal session configurations.

This paper is organized as follows. In Section II, we review the optimal session configuration problem (without capacity constraints) and show how it can be solved using the layered graph method. In Section III, we extend the problem to include capacity constraints and show that the problem is $\mathcal{NP}$-hard. In Section IV we introduce several heuristic algorithms and in Section V, we present simulation results showing that the best of the heuristics is able to provide excellent performance.

II. Configuring Sessions with Layered Networks

In the session configuration problem, we are given a graph that represents the network, with nodes representing routers or switches and edges representing communication links. Our objective is to select a path for a session involving two *terminals* s and t and one or more intermediate processing steps $r_1, \ldots, r_k$. For each processing step r_i, there is a set of routers R_i that contains potential candidates for performing that step. A *feasible path* from s to t is a path that includes at least one element from each of the sets $R_1, \ldots, R_k$, in the proper order. An optimal path is one of the least cost, where the cost of a path is the sum of the costs of the links on the path and the costs of the nodes that are selected for carrying out the processing steps.

The *layered network method* solves the session configuration problem by reformulating the problem in another space, where it can be solved as a conventional shortest path problem. The resulting shortest path can then be mapped back to a path in the original network graph, to produce the required session configuration.

We illustrate the method, focusing on the transformation that converts the network graph to a new graph called the "layered network". Let us consider a unicast session with a single processing step, where R is the set of candidate nodes that are capable of handling the processing step. We refer to such nodes as servers in this paper. For this session, we transform the original network graph into a "two layer" graph. The layered network G' includes two copies of the original network graph $G = (V, E)$. We refer to one copy as layer 0 and the other copy as layer 1. Also, for each node v in G, we denote the copy of the node in layer 0 as v_0 and the copy in layer 1, v_1. The cost of each edge in G is preserved in both layers. We complete the layered network G' by adding an inter-layer edge (r_0, r_1) for each server r in R. Here, the cost of each server r is applied to the new edge, (r_0, r_1). Figure 1 shows an example of a layered network.

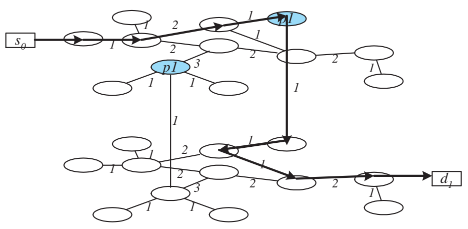


Fig. 1. Layered Network with Shortest Path

Given the layered network G', we compute the least cost path from the node s_0, (the copy of the source node s in layer 0), to the node t_1, (the copy of the destination node t in layer 1). Note that G' only has edge costs, so shortest path algorithms can be applied directly. Figure 1 shows the least cost path in the layered network.

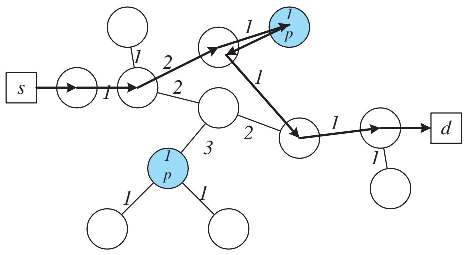


Fig. 2. Session Configuration

For the final solution to the unicast session configuration problem, the least cost path in G' is mapped back to the network graph G as follows. For each regular edge involved in the path, we "project" it to the original copy in G. Similarly for the inter-layer edge in the path, we "project" it to the original server in G and mark it as the designated node for the processing requested in the session graph. The projection yields a legitimate configuration connecting the two terminals and containing the server on the path. This configuration has the least cost among all such configurations (this is proved in [1]).

The layered network method can be generalized to an arbitrary number of processing steps. For a unicast that involves two terminals s and t and processing with k consecutive steps, we build the layered network with $k+1$ copies of the original network where the copies are denoted layer 0 through layer k. Between layer $i-1$ and layer i, we add an inter-layer edge (r_{i-1}, r_i) for each server $r \in R_i$, where R_i is the set of candidate servers for step i. Then, the optimal configuration is computed by finding the least cost path from s_0 to t_k in the layered network and projecting it back to the original network. Thorough descriptions and proofs with regard to the *layered network method* are given in [1].

III. Capacity Constrained Networks

Our original formulation of the session configuration problem does not take into account limits on the processing capacities of the nodes and bandwidths of links. Some applications require that a certain amount of link bandwidth and processing capacity be reserved for the application, in order to ensure that users experience an acceptable quality of service. To handle such applications properly, the session configuration algorithm must be extended to account for capacity constraints. For example, consider a video conferencing application involving intermediate processing steps to first compress the video, then decompress it. To ensure subjectively satisfactory performance, we need to ensure both that there is sufficient available bandwidth at every link on the path, and that the servers have sufficient processing capacity to perform the compression and decompression steps with minimal delay.

To account for capacity constraints, we must extend the session configuration problem to include capacity constraints for both edges and nodes in the network graph. We also add capacity requirements to the session model. That is, for each processing step, we specify a processing requirement, and for every path segment between two consecutive processing steps, we specify a bandwidth requirement. A feasible solution is one for which the demands placed on links and processing resources do not exceed the available capacity. As before, we seek a feasible solution of minimum overall cost.

Note that for applications that do not require intermediate processing, it's easy to find a minimum cost feasible solution. In this case we can simply remove from the network all links that lack the bandwidth needed for the session, and then find a shortest path in the reduced network. The introduction of intermediate processing steps complicates things, primarily because the best session configuration need not correspond to a simple path in the original network graph. In particular, there is a possibility that in the best configuration there are links and/or nodes that are used more than once. This makes it difficult to find the best configuration, while ensuring that each link and node is not over-used.

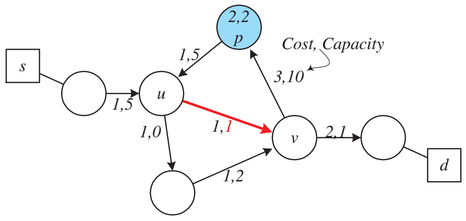


Fig. 3. Application with a capacity restriction

To illustrate this, consider the video transcoding example, in the network shown in Figure 3. In the figure, each link is labeled with a pair of numbers, the first being the cost of the link and the second being its available capacity. The shaded node p, is the only one capable of performing the transcoding required by the application. Note that if the bandwidth needed for the video is ≤ 0.5, there is a path from the source node s to the destination node d that passes through p and uses the edge (u, v) twice. However, if the bandwidth of the session exceeds 0.5, there is no feasible configuration. If we attempt to solve this problem using the layered network method, it's not clear what capacity should be assigned to edges on different layers. If we assign the original capacity to all copies, the subsequent shortest path computation could produce an infeasible path. If we divide the capacity among the different layers, we may prevent the discovery of some feasible paths.

The difficulty illustrated by this example is no accident. Indeed, the problem of optimally configuring a session with capacity restrictions is intractable. Consider a capacity-constrained network, $G = (V, E)$ in which every pair of nodes is joined by an edge, and where every node but s and t is capable of performing processing and has one unit of available capacity.

Now consider a session that requires $|V| - 2$ processing steps, each of which requires one unit of processing capacity. Any feasible solution to this problem must pass through all the intermediate nodes and thus, any feasible path provides a solution to the well-known Hamiltonian path problem [5], which is known to be $\mathcal{NP}$-complete. This argument shows that not only is it difficult to find the best solution, but it is difficult to find any feasible solution, even when all links have the same capacity and cost, and all servers have the same capacity and cost. Given the intrinsic intractability of the problem, we turn next to a study of efficient heuristic algorithms, which can be expected to have good performance in practice.

IV. Heuristic methods for capacity constrained networks

In this section, we introduce two heuristic methods for the optimal session configuration problem in a capacity-constrained network. We focus on unicast sessions that specify a processing requirement for each step and a bandwidth requirement for each path segment between two consecutive steps. Note that the bandwidth requirement can differ on different path segments, since processing steps may expand or reduce the amount of data. As described in [1], the costs in the different layers are scaled to account for such effects. Our heuristics extend the layered graph method to prevent resources from being used beyond their current capacity.

The first heuristic is really a collection of similar algorithms, that we refer to as *selective edge inclusion* algorithms. Each modifies the layered graph to prevent links from being over-used and then finds a shortest path in the modified graph. The algorithms differ in the way they modify the layered graph.

- The *strict inclusion* method includes an edge in the layered graph only if it has enough available capacity so that it cannot be over-used, even if it is selected for use in all layers. This policy applies to both intra-layer and inter-layer edges. Since different processing steps may require different amounts of processing capacity, we include a given edge as an inter-layer edge only if the sum of the capacities required for all the processing steps is no larger than the available capacity of the server represented by the inter-layer edges. Similarly, we include a given intra-layer edge only if its capacity is no smaller than the sum of the bandwidth requirements for all path segments. Once the modified layered network is constructed, a shortest path from the source to the destination is found. If none exists, the session configuration attempt is rejected.
- The *loose inclusion* method includes an edge in all layers if it has sufficient capacity to be used in any layer. If, after a path is determined, the path is found to over-use some edge, the path is discarded and the session configuration attempt is rejected.
- The *permissive loose inclusion* method is not intended as a practical algorithm, but is used in the simulation study to provide a nominal bound on the performance of the other algorithms. It works like the loose inclusion method, except that it never rejects the path that is found, even if the path over-uses some edge.

- The *random inclusion* method includes edges in a set of selected layers for which the total capacity is no larger than the edge capacity. For each edge, the layers are selected randomly and independently. Once the modified layered network is constructed, shortest path search is done. If successful, the session is configured using that path.
- The *consecutive inclusion* method picks a layer at random and then goes through the remaining layers in consecutive order, adding the edge to each layer in which the addition does not violate the capacity constraint.

The selective edge inclusion methods are very simple to implement and can perform reasonably well when the session resource requirements are much smaller than the capacities of the links and servers.

Our second heuristic is somewhat more complex but can perform well, even when session resource requirements are relatively large. The algorithm is an extension to Dijkstra's shortest path algorithm, and is called the *capacity tracking* algorithm. 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 implemented using a Fibonacci heap [6], 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

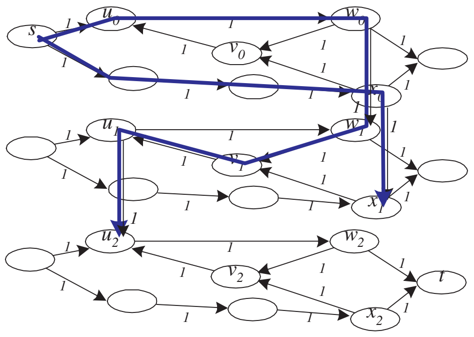


Fig. 4. Link capacity tracking (Blocked)

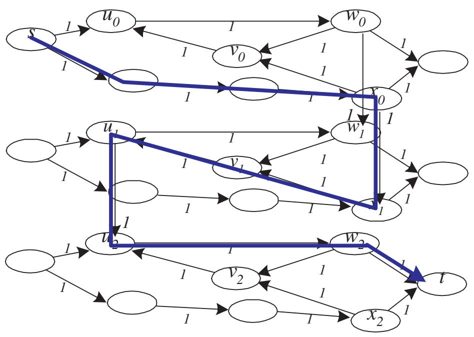


Fig. 5. A valid configuration

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 capacity checking procedure as *link capacity tracking*.

The extra time required by link 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.

Link 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 Figure 4. 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 Figure 4 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_2 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 Figure IV.

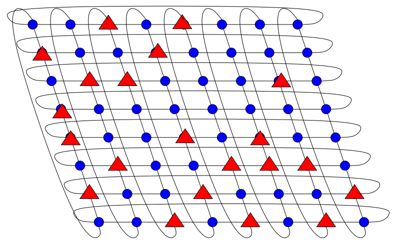


Fig. 6. Torus network

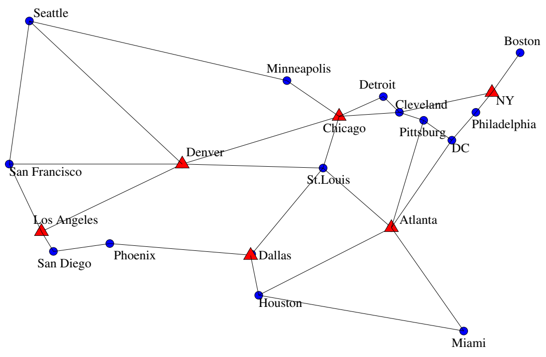


Fig. 7. Metro 20 Network

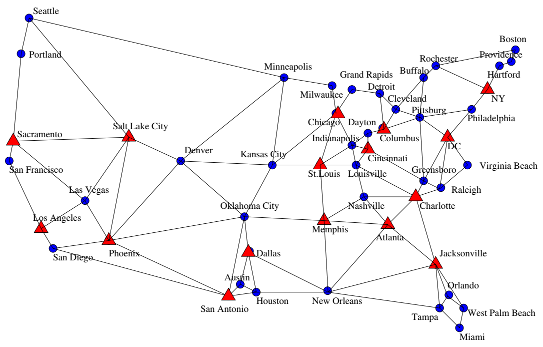


Fig. 8. Metro 50 Network

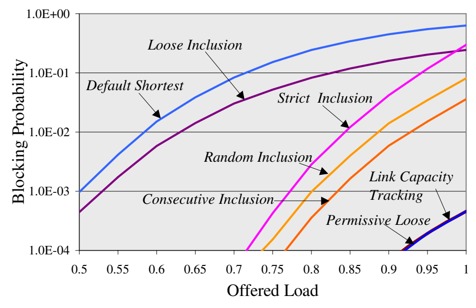
V. Simulation Results

We performed a set of simulations for the session configuration problem to evaluate the algorithms presented in the previous section. Simulations were performed using four different network topologies.

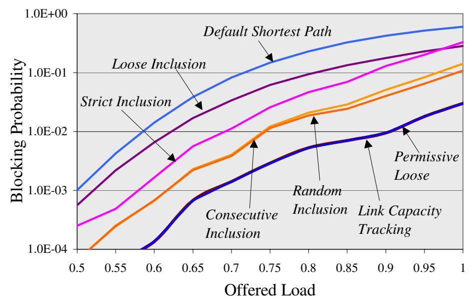
- *Torus*: This network is based on a grid of 64 nodes where every node has an outgoing edge to each of its four neighbors, north, south, east and west along the grid lines. The nodes at edges of the square grid also have links that "wrap around" to the corresponding node at the opposite edge, resulting in a torus topology. The network has 128 edges and each is assigned an equal cost and capacity. Figure 6 shows the network topology where servers are shown as triangles. A random subset of the nodes are designated as servers, with the ability to perform processing. All servers have the same capacity.
- *Random*: This network is a random regular network with 64 nodes, each having 4 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 Figure 7. 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. 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 servers along each default path are dimensioned to handle the worst-case traffic load allowed by the traffic constraints.
- *Metro 50*: is a larger version of the *Metro 20* network. It has a node for each of the fifty largest metropolitan areas in US. The topology is shown in Figure 8. The links and servers are dimensioned in the same way as *Metro 20*.

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.

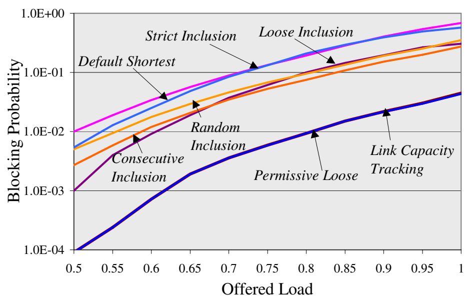
- *Density of servers* (P): The density of servers is just the ratio of the number of servers to the total number of nodes. In the results reported here $P = \frac{1}{3}$. The servers were randomly selected for *Torus* and *Random* and were configured for *Metro 20* and *Metro 50* as shown in Figure 7 and Figure 8, 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.



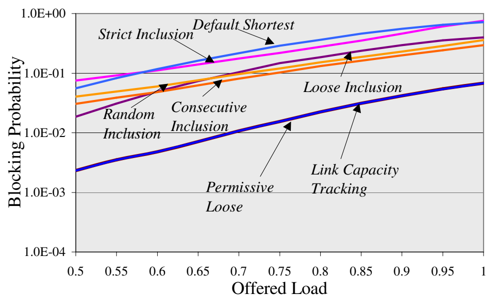
(a) Torus Network



(b) Random Network

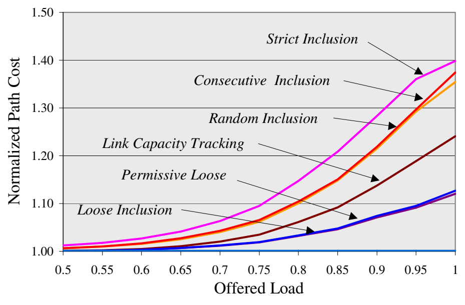


(c) Twenty metro areas

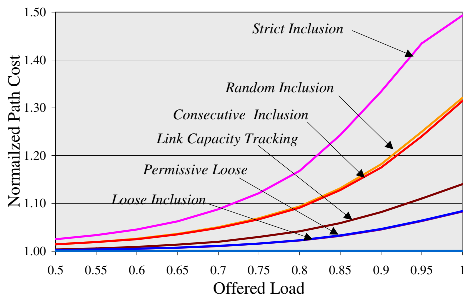


(d) Fifty metro areas

Fig. 9. Heuristics for session configurations



(a) Twenty metro areas



(b) Fifty metro areas

Fig. 10. Configuration Cost

- *Number of steps* (N_{steps}): The number of processing steps that a session requires.
- *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 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), then attempting to connect random pairs of nodes. Each simulation run included over 2.5 million session setup attempts.
- *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 to 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. Figure 9 shows the blocking probabilities for the various heuristics, as a function of the offered load. The plots also show the blocking probability when paths are constrained to use the default path. 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, *Link Capacity Tracking* outperformed the heuristics that use *Selective Inclusion*. Also, note that *Link Capacity Tracking* generally performs almost as well as the permissive loose inclusion method, which is included as an idealized bound on algorithm performance.

For *Torus*, every heuristic method except *Loose Inclusion* results in blocking probability less than 1% for load up to 80%. (See Figure 9(a).) Note that *Torus* has many paths between selected end nodes, and therefore, the heuristics that avoid overusing resources by ignoring some edges in the layered graph still have a good chance of finding valid paths in the reduced graph. On the other hand, *Loose Inclusion* does relatively poorly, apparently because it often selects paths that over-use resources (primarily servers).

For *Random*, all the better 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. With *Random* 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 tend 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 link capacity tracking, blocking probabilities of less than 1% are obtained for offered loads of more than 75%.

We also measured the cost of the successful configurations. In Figure 10, we show the configuration cost from all heuristics relative to the cost of the default shortest path, which is a lower bound. All heuristics provide nearly optimal costs at low loads, but deviate significantly at higher loads. The paths produced using *Link Capacity Tracking* generally stay within 5 to 10% of the lower bound up to loads of 95%. For the *Metro 20* network the cost rises to about 20% more than the lower bound at a load of 95%.

Lastly, we measured the average time required for session configuration by the different algorithms. Figure 11 shows the results for *Metro 50*. For all algorithms, we varied the number of steps from 1 to 10. As can be seen, the algorithms based on selective link inclusions are the fastest. On the other hand, link capacity tracking remains reasonably competitive, with a computational cost less than twice that of the best selective inclusion 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 *Link Capacity Tracking* more than compensates for the extra computational time.

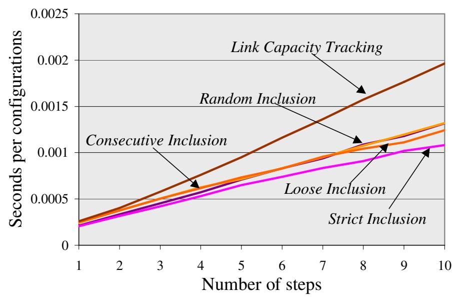


Fig. 11. Time requirements for session configurations

VI. Summary

Programmable network elements are expected to become more common, in the years ahead, enabling advanced network services that do more than simply transport bits from place to place. As the use of advanced services grows, it will become more important to understand how to configure application sessions to make best use of the available resources.

In this paper, we have presented a general approach to the problem of configuring application sessions that require intermediate processing, and that require reserved capacity. We have studied several heuristic algorithms for this problem, based on the layered graph method, including a novel extension of Dijkstra's shortest path algorithm, to account for capacity constraints. Our simulation results demonstrate that the *link capacity tracking* algorithm matches the best performance that one can expect to achieve.

References

[1] Sumi Y. Choi, Jonathan Turner, and Tilman Wolf, "Configuring session in programmable networks," *Proceedings of IEEE Infocom 2001*, Apr. 2001.

[2] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, Jan. 1997.

[3] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela, "A survey of programmable networks," *Computer Communication Review*, vol. 29, no. 2, pp. 7–23, Apr. 1999.

[4] Daniel Decasper, Guru Parulkar, Sumi Choi, John DeHart, Tilman Wolf, and Bernard Plattner, "A scalable, high performance active network node," *IEEE Network*, January/February 1999.

[5] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to NP –completeness*, Freeman, San Francisco, 1979.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, McGraw-Hill Book Company, 1990.

[7] Hongzhou Ma, Inderjeet Singh, and Jonathan Turner, "Constraint based design of atm networks, an experimental study," *Washington University Computer Science Department Technical Report WUCS-97-15*, 1997.