ON THE SELECTION OF NUMBERS OF SERVERS FOR
THE N SERVER-TYPE PROBLEM

by

David Mark Lampbell

ABSTRACT

A probabilistic model is developed for the evaluation of the cost
for a configuration of servers in an assembly tree structure of server-
types. A method of optimizing the number of servers of each type with the
objective of minimizing cost is presented.

A telecommunications application is considered and analyzed. Compu-
tational results are presented.
TABLE OF CONTENTS

0: Introduction

1: The N Server-Type Problem

II: Application of the N Server-Type Problem to the
WATSBOX Optimization Problem

Note to the Reader

Appendix A

Bibliography

Page

1
2
22
36
37
39

LIST OF TABLES

Table

2.1 WATS Line Charges for Ithaca, New York
2.2 Computational Results

Page

25
32
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Typical Cost Functions</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Potential Path of Overflow, 3 Server-types</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Potential Path of Overflow, 4 Server-types</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Postorder Labeling of the Assembly tree</td>
<td>10</td>
</tr>
<tr>
<td>1.5</td>
<td>Enumeration Tree</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>Assembly Tree for Cornell University</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>Initial Assembly Tree-Cornell University</td>
<td>31</td>
</tr>
<tr>
<td>2.3</td>
<td>Separation of the Assembly Tree</td>
<td>34</td>
</tr>
</tbody>
</table>
0: INTRODUCTION

Consider a system with an input stream of jobs each of which must be serviced with no more than a minor delay. In order to service the demand, our system has a set of server-types, allowing zero or more servers of any particular server-type. The cost of providing the servers is a complicated nonlinear function of the number of servers of each type and the load. Each type of server may be restricted from handling part of the input stream. This gives rise to a natural constraint, namely, each job must be serviced by a server capable of handling it. The objective is to pick the number of servers of each type which minimizes the cost of operating the system.

The problem arises naturally in the study of a telecommunications system. Results in this area have not covered the $N$ server-type problem with different servers at different costs capable of handling some of the jobs and with our objective [11].

Chapter I deals with the general problem. The servers are discussed, a model of the system is developed to calculate costs, and an optimization scheme is presented.

Chapter II consists of a telecommunications application, a WATSBOX, a computerized controller of outgoing telephone lines. This problem is shown to fit the general framework. We provide computational results based upon computer code for this application.
Chapter I

THE N SERVER-TYPE PROBLEM

We develop a solution to the N server-type problem. The problem statement is:

let \( N \) be the number of server types

\( L_i \) be the input load for server \( i, \ i=1,2,...,N \)

\( L \) be the vector \( (L_1, L_2, \ldots, L_N) \)

\( S \) be a vector of number of servers of each type

\( C_i(S,L) \) be the cost associated with all servers of type \( i \)

\[
\text{Min} \sum_{i=1}^{N} C_i(S,L)
\]

s.t. \( S_i \geq 0, \text{ integer } i=1,\ldots,N \)

We will study this problem by discussing the structure of the servers, by developing a computational method for evaluating \( C_i \), and lastly by considering the problem of optimization.

It is important to the problem that the structure of the servers be understood. Each server may handle only one job at a time. The difference in server types is that servers may be allowed to handle only certain jobs out of the input stream. This is called the GENERALITY of the server-type. The most general server-type is capable of handling any job in the load; the least general, only a small fraction of all possible jobs that could be handled. Note that this fraction does not deal with the actual number of jobs in the load. For example, if we had a load
which could contain a maximum of 10,000 different job types, the most
general server-type might be able to handle any of the 10,000 job types
while the least general server-type might handle only one particular job
type. That job type could represent 90% of the jobs in the load, however;
so the least general server-type might have a large "load". We assume
that there exists a server-type that can handle any of the potential
jobs. Each server, capable of handling a job, services it at the same
speed. For example say a job j can be handled by server-type i and
server-type k; a server-type i or server-type k server would handle
job j in the same amount of time.

Definition: The DIRECT LOAD of a server-type consists of all jobs
for which the server-type is the least general server
capable of handling those jobs.

In the problem statement (1), \( L_i \) is the direct load for server-type
i, for \( i=1,2,...,N \).

The cost of a server is a function of its use and generality. We
assume that the more general the server, the more it costs (compared to
a less general server at the same use level). Each server may have a flat
fee associated with it and a charge for the amount of use. In any case,
the cost of a server is assumed to be an increasing, but not always convex,
function of use. All servers of the same type have the same cost function.
We shall consider use as the amount of time the server is in operation.

Note that this cost function is not the \( C_i \) of (1). Figure 1.1 shows
typical cost functions for servers. Figure 1.1a is a less general server
than that of Figure 1.1b, while Figure 1.1c is a nonconvex cost function.
The servers obey the rule: the more general the server, the higher the
Typical Cost Functions

Figure 1.1a

Figure 1.1b

Figure 1.1c

Figure 1.1
cost of operation.

The most general server-type is assumed to have a linear cost function which passes through the origin. We may therefore, assume an infinite number of servers of this type.

Consider a job that can be served by a low generality server-type. If all servers of that type are busy, the job could be serviced by a more general server-type. This process could continue. In effect the job may travel up a chain of server-types. The potential path of a job may be represented by Figure 1.2.

Definition: There is OVERFLOW from server-type \( i \) when there are no free servers of type \( i \) available to handle a job requesting a type \( i \) server.

It is also possible that there exists another low generality server-type, say type \( 4 \), which may overflow into server-type \( 2 \) as the next server-type on the potential path of a job. In such a case we assume that the set of potential jobs serviced by server-type \( 1 \) and the set of potential jobs serviced by server-type \( 4 \) are disjoint. Such a situation is represented by Figure 1.3.

Definition [1]: A DIRECTED GRAPH \( G = (V,E) \) consists of a finite nonempty set of NODES \( V \) and a set of EDGES \( E \). Each edge is an ordered pair \((v,w)\) of vertices, \( v, w \in V \); \((v,w)\) is from \( v \) to \( w \).

A PATH in a directed graph is a sequence of edges of the form \((v_1,v_2), (v_2,v_3), \ldots, (v_{n-1},v_n)\).
Potential Path of Overflow, 3 Server-types

most general server-type

3

2

least general server-type

1

Figure 1.2

Potential Path of Overflow, 4 Server-types

most general server-type

3

2

least general server-types

1

4

Figure 1.3
The LENGTH of this path is \( n-1 \). A single node is a path of length 0.

A path is SIMPLE if all the edges and all the nodes on the path, with the possible exception of the first and last nodes, are distinct. A CYCLE is a simple path of length at least 1 with the first and last node the same.

A directed graph with no cycles satisfying the following properties is a (ASSEMBLY) TREE:

1) there is exactly one node, called the ROOT, from which no edge exits.

2) every node, except the root, has exactly one exiting edge,

3) there is a path from each node to the root.

Under our assumptions the overflow structure of the system is an assembly tree. That the structure is no more complex than an assembly tree may be seen by considering a server-type \( i \) with two edges leaving it. Those edges must join two other server-types, say \( j \) and \( k \). If either server-type \( j \) can handle type \( k \) jobs or server-type \( k \) can handle type \( j \) jobs then there is a redundant edge. Therefore, the set of potential jobs serviced by server-type \( j \) and the set of potential jobs serviced by server-type \( k \) are disjoint. Both server-type \( j \) and \( k \) handle type \( i \) jobs. This contradiction proves that a server-type has at most 1 exiting edge.
Definition [1]: Let \( T = (V,E) \) be a tree. If \((v,w) \in E\) then \(v\) is called the KID of \(w\), \(w\) is the PARENT of \(v\). If there is a path from \(v\) to \(w\) then \(v\) is a DESCENDENT of \(w\). Further, if \(v \neq w\) then \(v\) is a PROPER DESCENDENT of \(w\). A node \(v\) and all its descendents are called a SUBTREE of \(T\); the node \(v\) is called the ROOT of the subtree. If \(v\) and \(w\) are both kids of node \(y\) then \(v\) and \(w\) are SIBLINGS.

When convenient the assembly tree shall be referred to in tree notation of nodes, parents, kids, and siblings. A node is the same as a server-type. In cases where it is necessary to emphasize the reference to the assembly tree or for clarity, the abbreviation AT will be employed.

Definition: Let \( T \) be a tree with root \( r \) and kids \( v_1, v_2, \ldots, v_k \) of \( r \) \( k \geq 0 \). If \( k = 0 \) the tree consists of the single node \( r \). Let \( I = 1 \) initially. A POSTORDER labeling of \( T \) is defined recursively as:

1) visit in postorder the subtrees with roots \( v_1, \ldots, v_k \) in that order,

2) visit the root \( r \).

At a visit to a node \( i \): give node \( i \) the label \( I \) and then increment \( I \) by 1.

It is necessary to consider the overflow from all the proper descendents of node \( i \) before evaluating \( C_i(S,L) \). To this end all the server-
types will be named by the postorder labeling of the assembly tree. Figure 1.4 is the assembly tree of Figure 1.3 with postorder labeling.

The postorder labeling of the assembly tree simplifies the problem of evaluating the cost function $C_i$. The method employed is to calculate the overflow out of a node and add that overflow, which is in the form of unserviced demand, into the parent of the node. The postordering makes this possible. We shall shortly return to the question of the overflow.

The evaluation of $C_i(S,L)$ is based upon a birth and death process. It is assumed that the arrival of jobs in the direct load of each server-type follows a Poisson distribution. The service times are assumed to be exponentially distributed; the means may vary from type to type.

Strictly speaking, when the overflow from one server-type is added to the load of another server-type, the Poisson and exponential natures of the load are destroyed. Nonetheless, we shall treat the total load of each server-type as though the arrivals were Poisson and the service times were exponentially distributed. If all means of service times were equal, the distribution of service times in the total load would be exponential; if the means are nearly equal, as we shall assume they are, the distribution in question should be nearly exponential. At least in extreme cases the Poisson assumption is nearly valid: if there are few servers of a particular type then the overflow is nearly Poisson; on the other hand if there are many servers the overflow is nearly zero. In either case if the overflow is added to a Poisson process, the sum will be nearly Poisson.
Postorder Labeling of the Assembly Tree

most general server-type

3

least general server-types

1  2

Figure 1.4
For the evaluation of $C_1(S_i, L)$ consider the birth and death chain of a Poisson process with birth parameter $\lambda$ and death parameter $\mu$.

Let $\lambda$ be the arrival rate at node $i$, i.e., #jobs/time unit
$\mu$ be $1/(\text{average service time at node } i)$
t be the number of servers of node $i$.

The birth and death chain has $t+1$ states, $0, 1, \ldots, t$

let
$\mu_x = x \cdot \mu \quad x = 0, 1, \ldots, t$
$\lambda_x = \lambda \quad x = 0, 1, \ldots, t.$

This process is clearly irreducible and positive recurrent. The stationary distribution for a Poisson birth and death process is given by [8]:

$$
\Pi(x) = \frac{\Pi(x)}{t} = \frac{\sum_{k=0}^{t} \Pi_0}{t}
$$

where

$$
\Pi_x = \begin{cases} 
1 & x = 0 \\
\frac{\lambda_0 \cdots \lambda_{x-1}}{\mu_1 \cdots \mu_x} & x = 1, 2, \ldots, t.
\end{cases}
$$

The expected overflow from the node is calculated as follows:

$$
P\{\text{a job overflows in a small time unit} \} = P\{\text{being in state } t\} \cdot P\{\text{job arrives in a small time unit}\}
$$

$$
E[\text{overflow}] = \frac{1}{\mu} \cdot P\{\text{a job overflows}\}
$$

$$
= P\{\text{being in state } t\} \cdot \lambda/\mu
$$
\[ P(\text{being in state } t) = \Pi(t) \]
\[ = \frac{\Pi_t}{\sum_{k=0}^{t} \Pi_k} \]
\[ \Pi_x = \left(\frac{\lambda}{\mu}\right)^x \frac{1}{x!} \quad x=0,1,\ldots,t \]
\[ \Pi(t) = \left(\frac{\lambda}{\mu}\right)^t \left[ t! \sum_{k=0}^{t} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} \right]^{-1} \]
Hence,
\[ E[\text{overflow}] = \left(\frac{\lambda}{\mu}\right)^t \left[ t! \sum_{k=0}^{t} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} \right]^{-1} \frac{\lambda}{\mu} \]
\[ = \left(\frac{\lambda}{\mu}\right)^{t+1} \left[ t! \sum_{k=0}^{t} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} \right]^{-1} \]
and where \( \lambda/\mu \) is the input load.

The expected value of the load absorbed is given by
\[ E[\text{absorbed}] = \frac{\lambda}{\mu} - E[\text{overflow}] \]

This can now be used with the cost function of the node to determine \( C_i(S,L) \).

Notice that \( \frac{\lambda}{\mu} \), the load, must be adjusted to include the unserved jobs from the lower levels of the assembly tree.

The probability that the jth server of a node is in use can be easily determined. Consider the overflow of the first j-1 servers and the overflow of the first j servers. The difference is the amount of load absorbed by server j. Further as only one job from the load may be handled by server j at a time, we can say
\[ P[\text{server } j \text{ is in use}] = \left( \frac{\lambda}{\mu} \right)^j \left( \frac{\lambda}{\mu} - 1 \right) \sum_{k=0}^{j-1} \frac{\lambda^k}{\mu^k} \frac{1}{k!} - \frac{\lambda^{j+1}}{\mu^{j+1}} \left( \frac{\lambda}{\mu} \right)^j \sum_{k=0}^{j} \frac{\lambda^k}{\mu^k} \frac{1}{k!} \right]. \]

The evaluation of \( C_N(S,L) \), the cost associated with the root of the assembly tree, is considered separately. As node \( N \) is the most general server, the cost of operating an individual server may also depend upon what the job's minimum server was. Further, it may not be interesting to calculate overflow from this node but rather assume a large number of these servers. This is possible if the fixed costs are small but the use rates are large.

It is desirable to attribute the load into node \( N \) back to the original nodes for the evaluation of \( C_N(S,L) \). It has been shown [8] that the overflow from a node has the same composition as the input load (see Appendix A). Based upon this, we present a two-step procedure to attribute load back to the original nodes. First associate with each node \( i \) the sum of the overflow serviced by node \( N \) for all the descendents of node \( i \). Second, associate with each node \( i \) the overflow serviced by node \( N \) from node \( i \).

Let \( \text{OVERFLOW} (1:N) \) be the overflow from each node. Notice \( \text{OVERFLOW} (N) = 0 \).

\( \text{TLOAD} (1:N) \) be the load actually seen at each node. This is the direct load plus any overflow to the nodes.

\( \text{LOAD} (1:N) \) be the direct load of each node.

\( \text{OVER} (1:N) \) be the load of node \( N \) attributed to each node.

\( \text{PARENT} (1:N-1) \) be the parent of the node. I.E. \( \text{PARENT} (i) = j \) if \( j \) is the parent of \( i \).
The procedure is:

Step 0: NODE + N
Step 1: NODE + NODE - 1
    if NODE = 0 then go to Step 3
Step 2: if TLOAD(PARENT(NODE)) \neq 0
    then OVER(NODE) = OVER(PARENT(NODE)) \times TLOAD(PARENT(NODE))
    else OVER(NODE) = 0
    in either case go to Step 1
Step 3: NODE = 0
Step 4: NODE + NODE + 1
    if NODE = N then go to Step 6
Step 5: if TLOAD(NODE) \neq 0
    then OVER(NODE) = \frac{LOAD(NODE)}{TLOAD(NODE)} \times OVER(NODE)
    else OVER(NODE) = 0
    in either case go to Step 4
Step 6: terminate.

The vector OVER is the load handled by node N attributed to the originating nodes. Given this, the cost function of a server of node N can be evaluated and hence \( C_N(S,L) \) can be evaluated.

Definition: The ENUMERATION TREE for our problem is a tree in which each node is a possible configuration of servers. The tree contains those configurations which were considered while searching for the optimal configuration.
The nodes of the enumeration tree, for our problem, will correspond to a lexicographic ordering of the configurations. Figure 1.5 is the general form of the enumeration tree. Reference to the enumeration tree will be by ET where required.

Let $T_{S_1S_2\ldots S_{N-1}}^U$ be a node of the enumeration tree.

The subscript $S_1S_2\ldots S_{N-1}$ is an ordered vector which is the configuration at the node. The superscript $U$ indicates the maximum server-type that can be altered when constructing descendents. The use of the superscript prevents a configuration from being considered more than once.

Only the addition of servers will be considered. As an example consider the node $T_{1101}^2$; only servers of server-type 1 or server-type 2 may be added. Since it is assumed that node $N$ has many servers, it is sufficient to consider nodes 1 to $N-1$.

let $\epsilon_k$ be the vector $0\ldots010\ldots0$ where the 1 is in the $k^{th}$ position.

The separation [4] of a node $T_S^j$, $1 \leq j \leq N-1$ is create the new nodes $T_{S+i\epsilon_j}^1$, $i=1,\ldots,j$ and add them to the enumeration tree.

The order of evaluation of nodes is the rightmost and deepest node has priority. Whenever any choice remains, pick the node $T_S^j$ such that $j$ is as close to $N-1$ as possible and $S$ has the most number of servers.

At each node, a feasible solution $Z(T_S^j)$, to the problem is produced and its cost is available. From these the incumbent solution, $Z_0$, is determined.
Enumeration Tree

Figure 1.5
The bound used for fathoming \([4]\) is related to \(C_N(S,L)\). At some node \(T_S^j\) only server types \(1\) to \(j\) may be increased. Let \(P\) be the cost attributed to server types \(1\) to \(j\) by \(C_N(S,L)\). If \(Z(T_S^j) - P > Z_0\) then \(T_S^j\) may be fathomed. In this case, the costs related to all servers less the maximum the cost of the system can be reduced is no better than the incumbent solution. A better bound may be achieved by using \(P'\) where \(P' = P - \min_{i=1,...,j} \text{ flat fee of server } (i)\). The flat fee of server \(i\) was explained in the discussion of the servers.

The branch and bound algorithm:

Step 1: start at \(T_{0...0}^{N-1}\)
- incumbent solution + \((0,...,0)\)
- \(Z_0 + Z(T_{0...0}^{N-1})\)

Step 2: if no live ET nodes exist go to Step 6
- otherwise select live vertex \(T_S^j\) and go to Step 3

Step 3: if \(Z_0 > Z(T_S^j)\)
- then \(Z_0 + Z(T_S^j)\), incumbent solution + \(S\) and go to Step 5
- go to Step 4

Step 4: if \(Z(T_S^j) = P > Z_0\)
- then fathom \(T_S^j\) and go to Step 2,
- otherwise go to Step 5

Step 5: "separate \(T_S^j\)"
- form new ET nodes

\[ T_S^i + \epsilon_i \quad i=1,...,j \]

- go to Step 2.
Step 6: terminate, optimal solution is the incumbent solution and the optimal value is $Z_0$. Improvements may be made to this algorithm. In Step 1, $Z_0$ may be set to the cost of any known solution to the problem. The incumbent solution, the best solution discovered so far, is initialized to the configuration which yielded $Z_0$.

The following heuristic is based on a dynamic programming approach to the problem. Consider adding AT nodes and their respective direct load one at a time in reverse postorder. Let $P(i)$ be the path between node $i$ and node $N$ not including $N$. As each AT node $i$ is added, calculate the maximum number of servers, say $k$, that can be added anywhere in the current assembly tree. Try adding, in order, $0$ to $k$ servers distributed on $P(i)$ in all possible manners to the best current configuration, and evaluate the cost. Pick the best cost and the configuration that yielded it. Increase the assembly tree by another node and add servers to the configuration just picked. Continue until all AT nodes have been included in the tree.

The maximum number of servers that can be added is restricted by the cost of servicing the direct load via node $N$. The cost of the direct load and overflow of existent assembly tree serviced by node $N$ divided by the server cost of node $i$ is a bound on the number of servers that need be tried.

The process of distributing servers to nodes can be thought of as placing $r$ balls in $m$ boxes. All possibilities must be tried implicitly. Whenever the cost of the servers is not less than the cost of the best solution to date, possible configurations may be ignored. The process of
adding servers may be terminated when a configuration is evaluated in which the cost of the servers is not less than the cost of the best solution to date and there is a node \( j \neq i \) on \( P(i) \) such that no trial servers have been allocated to nodes on \( P(j) \). This termination is based upon the idea that more general servers are more expensive and an allocation method that tries less general server nodes first.

The heuristic is:

let \( L_j \) be the vector \((0,0,\ldots,0)\), direct load node \( j,0,\ldots,0 \) \( j=1,\ldots,N-1 \)

Step 0: \( i \leftarrow N \)

\[ S \leftarrow (0,\ldots,0) \]

\[ L \leftarrow 0_N \quad \text{where } 0_N \text{ is the zero vector of } N \text{ elements} \]

Step 1: \( i \leftarrow i-1 \)

if \( i < 1 \) then go to Step 3.

Step 2: "optimally add lines to handle additional load"

\[ S \leftarrow S + A \]

where \( A \) is the allocation of additional servers picked by

\[ D = \min_{\text{feasible } A} \left\{ \sum_{j=1}^{N} C_i(S+A,L+L_j) \right\} \]

where feasible \( A \) has been described above,

go to Step 1

Step 3: terminate with configuration \( S \) at cost \( D \).

The nonoptimality of the heuristic is demonstrated by a simple counterexample. Consider on \( N=3 \) problem with the assembly tree of Figure 1.2.
The cost structure is:

<table>
<thead>
<tr>
<th></th>
<th>node 1</th>
<th>node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_N$ cost</td>
<td>$1200/period</td>
<td>$3600/period</td>
</tr>
<tr>
<td>Server costs</td>
<td>$1750/period</td>
<td>$1799/period</td>
</tr>
<tr>
<td>Direct load</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

at node 2

maximum #servers: \[\frac{3500 \times 1}{1799} \Rightarrow 2\text{ servers}\]

configuration

X0:

\[3600\]

X1:

\[1799 + \frac{1}{2} (3600) = 1799 + 1800 = 3599\]

X2:

\[3598 + \frac{1^3}{2(1+1+\frac{1}{2})} (3600) = 3598 + \frac{1}{5} (3600) > 3599\]

so best configuration is X1,

at node 1

maximum #servers: \[\frac{1800+1200}{1750} = \frac{3000}{1750} \Rightarrow 1\text{ server}\]

configuration

01:

\[1799 + \frac{2,2}{1(1+2)} \left[ \frac{1}{2} (1200) + \frac{1}{2} (3600) \right] = \]

\[1799 + \frac{2}{3} \times 4800 = 1799 + 3200 = 4999\]

11:

\[1750 + 1799 + \frac{(3)^2}{1+\frac{3}{2}} \left[ \frac{1}{3} (1200) + \frac{2}{3} (3600) \right] = \]

\[3549 + \frac{3}{4} \times \frac{2}{5} [1200+7200] = 3549 + \frac{3}{10} [8400] = \]

\[3549 + 2520 = 6069\]
\[ 3598 + \frac{2^3}{2(1+2+2)} [2400] = 3598 + \frac{4}{5} [2400] = \]

\[ 3598 + 1920 = 5518. \]

So the heuristic produces an answer of $4999. Consider the configuration $(0,0)$; its cost is $4800. Therefore the heuristic did not produce the optimal answer.

Another counterexample with $N=4$ can be constructed to demonstrate the nonoptimality of the heuristic. In this example node 3 servers are added when node 2 is added. Upon adding node 1, the optimal configuration has no node 1 servers. Due to the large number of calculations, the example is not presented.

At this point the general $N$ server-type problem is solved. An optimal solution may be found by the branch and bound algorithm or a computationally less costly but possibly nonoptimal solution may be found by the heuristic.

In the next chapter, an application of the $N$ server-type problem is considered. Computational experience will be related.
Chapter II
APPLICATION OF THE N SERVER-TYPE PROBLEM
TO THE WATSBOX OPTIMIZATION PROBLEM

The telephone company offers several types of long distance voice
grade lines which can often be employed to reduce long distance direct dial
station-to-station charges. Since a call, while placed on only one line,
may have the potential of being handled by more than one line type,
a control mechanism is required to match calls to lines. The traditional
controller is an operator; however a computerized controller, known as
a WATSBOX is commercially available from ACTION COMMUNICATIONS INC. The
problem of selecting type and number of lines with the objective of
minimizing long distance costs while under the control of a WATSBOX is
an N server-type problem.

Several types of voice grade lines are available from the telephone
company. We limit our attention to Wide Area Telephone Service (WATS)
lines, Foreign Exchange (FX) lines, and, of course, direct distance dial
(DDD) lines. Other line types, such as tie lines which connect private
switching equipment, could be considered within our framework.

The most general purpose line is a DDD line. This is just a local
calling area telephone line upon which long distance calls may be placed.

All the other line types that we consider may only handle calls that
could have been sent direct dial station-to-station with no operator
assistance. We shall consider calls or traffic of this nature only.
WATS lines are the next most general lines. They are available in 6 bands, where band refers to calling area. Band 0 can only handle intrastate traffic. Band 1 to band 5 can handle calls to various geographic areas. Band 1 can handle traffic to the states neighboring the user's state; band 2 can handle the band 1 traffic and additional calls, etc. Band 5 is capable of handling calls to anywhere in the continental U.S.A., however, band 1 to band 5 may not handle intrastate calls. Succinctly, in set notation:

\[ \text{let } C_i = \text{set of calls servicable by WATS band } i, \ i=0,1,\ldots,5 \]
\[ U = \text{set of station-to-station calls in continental U.S.A.} \]

then

\[ U = C_0 \cup C_5, \]
\[ C_5 \cap C_0 = \emptyset, \text{ and} \]
\[ C_5 \supset C_4 \supset C_3 \supset C_2 \supset C_1. \]

There are two types of WATS lines, full time and measured time. Measured time WATS lines are available for all WATS bands; full time WATS lines are available for WATS bands 1 to 5. The AT&T timing of WATS calls is the same as the timing of DDD calls [5].

A full time WATS line is available for 240 hours of use per month for an initial fee. A charge, based upon use, is imposed for any usage above the initial hour allotment. In many cases the initial allotment of 240 hours per month will not be exceeded. For any month, there are usually no more than 23 high use days. At 8 hours of use per day only 184 of the 240 hours are utilized.
Measured time WATS lines are charged according to usage. An hourly rate is imposed for use above the small initial hour allowance which is included in an initial fee. The hourly rate may be a function of the number of hours the line is used. For example, as of July 1976 for Ithaca, New York, WATS band 1 to WATS band 5 measured time lines have 10 hours initially and a constant rate per hour for all excess hours. The WATS band 0 measured time line rate is [5]:

<table>
<thead>
<tr>
<th>Hours</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>first 10 hours</td>
<td>$300/month</td>
</tr>
<tr>
<td>next 35 hours</td>
<td>$23.40/hour</td>
</tr>
<tr>
<td>next 35 hours</td>
<td>$8.00/hour</td>
</tr>
<tr>
<td>next 40 hours</td>
<td>$4.50/hour</td>
</tr>
<tr>
<td>over 120 hours</td>
<td>$2.00/hour</td>
</tr>
</tbody>
</table>

Notice that the WATS band 0 cost function is not convex (see Figure 1.1c). Table 2.1 contains charges for the other WATS bands.

A Foreign Exchange (FX) line allows a telephone user to have a telephone which is a local phone in a distant calling area. For example, in Ithaca, New York a person could have a local telephone in Washington, D.C. The tariff on an FX line is the monthly rental on the line to the distant area (line rental) plus local to the FX area charges and any long distance charges for calls dialed out of the FX area. The line rental is fixed and does not depend on the amount of use.

As would be expected, generality of the server is related to the cost of the line measured in some standard time unit such as minutes. FX lines are the least general; DDD lines the most general.
WATS Line Charges For Ithaca, New York

<table>
<thead>
<tr>
<th>WATS band</th>
<th>FULL TIME</th>
<th>MEASURED TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1st 240 Hours ($/month)</td>
<td>Each Additional Hour ($/month)</td>
</tr>
<tr>
<td>WATS band 1</td>
<td>1150.00</td>
<td>3.19</td>
</tr>
<tr>
<td>WATS band 2</td>
<td>1570.00</td>
<td>4.36</td>
</tr>
<tr>
<td>WATS band 3</td>
<td>1630.00</td>
<td>4.52</td>
</tr>
<tr>
<td>WATS band 4</td>
<td>1655.00</td>
<td>4.59</td>
</tr>
<tr>
<td>WATS band 5</td>
<td>1675.00</td>
<td>4.65</td>
</tr>
</tbody>
</table>

WATS band 0 see page 24.

Rates in effect, July 1976.

TABLE 2.1
All the assumptions of the N server-type problem are satisfied. DDD, WATS bands, and FX areas are the server-types. The line types form an assembly tree (for an example see Figure 2.1). DDD is the root of the tree, to evaluate its cost function one must attribute calls back to the originating bands.

The WATSBOX is designed to place telephone calls upon outgoing lines. It must match calls to required lines and place the call. As a by-product, access codes of users are checked for validity and a transaction record is produced for each call.

The hardware of the WATSBOX includes: a set of input ports; a set of output ports; a switching network between the input ports and the output ports; and a small computer with disk drive, tape transport and teletype.

For each call the WATSBOX executes several steps. The call is answered by an input port. The access code of the user, which the user dials, is checked by the computer. The caller next dials the desired telephone number. The computer determines the least general server-type that can handle the job. If it or any more general server except DDD is available the call is placed on the selected line. To place a call, the WATSBOX redials the number via touchtone dialing signals and then connects the input port with the output port. Should there be no line available to handle the call, the call will be placed in a short term queue. In the event that a queued call is not placed before the maximum retention time in the queue is reached, the call is placed upon a DDD line. Free long distance calls, calls requiring operator assistance, or calls outside the continental U.S.A. are routed to DDD immediately. A transaction log, which includes number dialed, date, time the call is placed, length of the
call, etc. is produced on magnetic tape.

The WATSBOX matches calls to servers by moving up the assembly tree. WATS band server-types may include both full time and measured time lines; the full time lines are tried first. There will be further consideration of this in the WATSBOX optimization discussion. A queue is imposed before DDD lines are tried.

A facility, called the flexible matrix, is available for the WATSBOX and allows implementation of other means for matching calls to servers. For each server-type, a row of the matrix specifies a sequence of server-types and the queue location, if used, that is the server-types overflow path to DDD. The full time and measured time lines of a WATS band may be separated and the queue, which is optional, may be placed anywhere on the path. We will not consider the problem of selecting the best allocation algorithm but will use the algorithm that comes with the WATSBOX as installed at Cornell University.

The purpose of the WATSBOX's queue is to place more calls on the cheaper servers, i.e. WATS and FX lines. This can be accomplished by presenting calls to the lines as soon as the line is free; consequently the line has less idle time. For this to occur calls must be in the queue. Calls get into the queue only when the system is busy and calls arrive at a high rate relative to the rate at which they leave the system. Under such conditions the lines would have little idle time in the absence of the queue; consequently, the number of calls placed DDD should be at most only slightly reduced given the queuing feature. When the system is operating under light load, the lines are free and calls will not get into the queue. The work of Buten and Doherty [2] indicate the savings due to the queue was less than 1% of the total expense of the system. We feel
it is reasonable to ignore the queuing feature, and do so.

The WATSBOX times the length of every call it places. The length of a call, as timed by the WATSBOX, starts when the call is placed; it includes the ring time. An unanswered call requires the use of a line and that time is in the call length. AT&T timing of a call begins when the call is answered and is rounded up to the next higher minute if 16 seconds or more, otherwise is rounded down. The WATSBOX timing is useful for line utilization considerations while AT&T timing is the time used for billing purposes.

The WATSBOX problem fits within the general probabilistic model with only minor adjustments. The timing of the model is based upon line utilization but the model calculates cost which requires AT&T timing. An adjustment of WATSBOX time to AT&T time needs to be applied.

The WATSBOX placement of calls on WATS band by first trying the full time then measured time lines may be handled as follows:

Let \( f \) = the number of full time lines of the band,
\( m \) = the number of measured time lines of the band,
\( \text{over} \) = the expected overflow from the \( f + m \) lines,
\( \text{fixedo} \) = the expected overflow from \( f \) lines.

Then
\[
E[\text{load absorbed by } ft] = \text{load} - \text{fixedo},
\]
\[
E[\text{load absorbed by } mt] = \text{fixedo} - \text{over}.
\]

The load handled by DDD, the \( N^{th} \) server-type, needs to be attributed to the server-types for the cost calculations.
We broke the month into two day types: high use and low use days. Each day was segmented by hour. The load was divided similarly and applied to calculate the expected cost of the system. Line utilization in hours could be accumulated over these divisions but the cost of DDD had to be calculated for each day type and time period.

Statistics were gathered for the month from the DDD toll tape available from the telephone company and from the WATSBOX transaction tape. The statistics gathered were: number of calls broken by day type, hour, and least general server-type; average length of calls by day type, hour, and least general server-type; average DDD costs per minute broken by day/evening and server-type; and the number of high use/low use days. The ratio between WATSBOX timing and AT&T timing figures is necessary for use as an adjustment in the cost calculations.

The computational results of our model were quite good. We analyzed the system of Figure 2.2 for April-May telephone load of Cornell University. Table 2.2 contains the results. The actual utilization of the lines was not, in its entirety, available for April-May. Differences may be explained by: 1) the actual results are an instance of a random variable as opposed to the expected value which we calculate; 2) the number of days do not match between the actual and the model; 3) there may be setup time between calls on the lines that is not reflected in the model; and 4) the queuing feature is ignored by the model but was in effect in the actual system.
Figure 2.2
### Computational Results

<table>
<thead>
<tr>
<th></th>
<th>Actual usage April 1 - April 28 20 high use days: hours:minutes</th>
<th>Actual usage April 13 - April 28 May 3 - May 12 20 high use days: hours:minutes</th>
<th>Model Results April 13 - May 12 23 high use days:...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WATS band 1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>full time</td>
<td>380:08</td>
<td>394:23</td>
<td>461.66</td>
</tr>
<tr>
<td>measured time</td>
<td>100:34</td>
<td>108:30</td>
<td>110.39</td>
</tr>
<tr>
<td><strong>WATS band 2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>full time</td>
<td>285:14</td>
<td>307:00</td>
<td>338.88</td>
</tr>
<tr>
<td>measured time</td>
<td>47:23</td>
<td>47:26</td>
<td>45.51</td>
</tr>
<tr>
<td><strong>WATS band 3</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>full time</td>
<td>206:33</td>
<td>209:20</td>
<td>233.10</td>
</tr>
<tr>
<td>measured time</td>
<td>49:59</td>
<td>49:05</td>
<td>46.22</td>
</tr>
<tr>
<td><strong>WATS band 4</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>full time</td>
<td>81:43</td>
<td>83:06</td>
<td>89.37</td>
</tr>
<tr>
<td><strong>WATS band 5</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>full time</td>
<td>299:18</td>
<td>306:18</td>
<td>321.36</td>
</tr>
<tr>
<td>measured time</td>
<td>1260:02</td>
<td>not available</td>
<td>1366.70</td>
</tr>
</tbody>
</table>

The number of days is different column to column.

**TABLE 2.2**
The model was implemented in PL/I and run on an IBM 370/168. A cost calculation for a configuration required 0.025 seconds of CPU time.

Buten and Doherty [2] consider a WATSBOX system for Proctor and Gamble, Cincinnati. They implemented a simulation model for the cost calculation which required 40 seconds of CPU time on an IBM 370/168 for the simulation of one day's performance. They did not make the exponential service time assumption or Poisson arrival assumption.

The selection of the optimal configuration for the WATSBOX was facilitated by a modification to the general solution techniques. Each of the nodes which overflows into DDD and all the node's descendents form a tree which may be treated as a separate problem. This has the effect of allowing us to solve smaller problems. Figure 2.3 shows the tree of Figure 2.1 separated into independent trees.

The problem of full time and measured time WATS lines was handled by ignoring the measured time line possibility if a full time line existed. Both the Buten and Doherty study [2] and the Klitz and Mecklenburg study [12] followed this approach. It is also our feeling that measured time lines, if used, should be employed to capture calls from DDD.

Both the heuristic and the branch and bound algorithm with the modification were used to optimize the system in Figure 2.1 for Cornell University telephone data. The heuristic generated its answer, which happened to be optimal, in 6.02 seconds of CPU time. The branch and bound algorithm performed 5188 cost calculations in 104.41 seconds of CPU time.

After the branch and bound algorithm selected the optimal answer, we added WATS band 5 measured time lines and selected the best resulting configuration. This had the effect of capturing some of the DDD bound calls on a less expensive server, but can also have the effect of altering
Separation of the Assembly Tree

Figure 2.3
what is the optimal configuration. We ignore this latter effect and just use band 5 measured time after the "optimization" is performed.

The following table summarizes the result of the WATSBOX optimization on Cornell April-May data:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial configuration</td>
<td>$39,891</td>
</tr>
<tr>
<td>optimization no FX</td>
<td>$35,486</td>
</tr>
<tr>
<td>optimization with FX</td>
<td>$30,383</td>
</tr>
</tbody>
</table>

We feel the general N server-type problem provided a good solution technique for the WATSBOX problem.

NOTE TO THE READER

This system was implemented in PL/I and required 5000 lines of code. Many details of the programming not germane to this thesis have not been included here.
APPENDIX A

"Proof" that output load has some "composition" as input load [7].

Assume: a server-type \( k \) has \( m \) input streams:

jobs of type \( i \); arrive in Poisson fashion, have arrival rate \( \lambda_i \),
and death rate \( \mu_i = 1/(\text{average service time for job of type } i) \).

\[ i=1, \ldots, m. \]

Assume: type \( i, i=1, \ldots, m \) arrivals are independent:

Overflow from server-type \( k \) at rate \( \lambda_0 \).

Assume: Input jobs are indistinguishable to the box, so that if
\( \lambda^{(i)}_0 \), is the rates of type \( i \) jobs overflowing from server-type
\( k, i=1, \ldots, m, \) then

\[
\lambda^{(i)}_0 = \frac{\lambda_i}{m} \sum_{j=1}^{m} \lambda_j, \quad i=1, \ldots, m.
\]

Assume: That the average service time of an overflow job, is the
same as the average service time of an input job; i.e.,

\[
\frac{1}{\mu_0} = \frac{\sum_{i=1}^{m} \lambda_i}{\sum_{i=1}^{m} \mu_i}
\]
The output load is given by

\[ L_0 = \lambda_0 / \mu_0. \]

The output load of type 1 calls is

\[ L_0^{(1)} = \lambda_0^{(1)} / \mu_1. \]

Now

\[
\frac{L_0^{(1)}}{L_0} = \frac{\lambda_0^{(1)} / \mu_1}{\lambda_0 / \mu_0}
\]

\[
= \frac{\sum_{i=1}^{m} \frac{\lambda_i}{\mu_i} \lambda_0 / \mu_1}{\lambda_0 \sum_{i=1}^{m} \frac{\lambda_i}{\mu_i}}
\]

\[
= \frac{\sum_{i=1}^{m} \lambda_i / \mu_i}{m \sum_{i=1}^{m} \lambda_i / \mu_i}
\]

\[
= \frac{\lambda_i / \mu_i}{m \sum_{i=1}^{m} \lambda_i / \mu_i}
\]

letting \( L_i = \lambda_i / \mu_i, i=1, \ldots, m \)

\[
\frac{L_0^{(1)}}{L_0} = \frac{L_1}{m \sum_{i=1}^{m} L_i}.
\]
BIBLIOGRAPHY


