Evaluating the Scalability of Distributed Systems

Prasad Jogalekar
Luminous Networks, San Jose, California

Murray Woodside *
Dept. of Systems and Computer Engineering
Carleton University, Ottawa, K1S 5B6 Canada
email: cmw@sce.carleton.ca

March 2000

Abstract

Many distributed systems must be scalable, meaning that they must be economically deployable in a wide range of sizes and configurations. This paper presents a scalability metric based on cost-effectiveness, where the effectiveness is a function of the system's throughput and its quality of service. It is part of a framework which also includes a *scaling strategy* for introducing changes as a function of a *scale factor*, and an *automated virtual design optimization* at each scale factor. This is an adaptation of concepts for scalability measures in parallel computing. Scalability is measured by the range of scale factors that give a satisfactory value of the metric, and good scalability is a joint property of the initial design and the scaling strategy. The results give insight into the scaling capacity of the designs, and into how to improve the design. A rapid simple bound on the metric is also described.

The metric is demonstrated in this work by applying it to some well-known idealized systems, and to real prototypes of communications software.

1 Introduction

Many distributed systems must be scalable. Typical present and future applications include web-based applications, e-commerce, multimedia news services, distance learning, remote medi- cine, enterprise management, and network management. They should be deployable in a wide range of scales, in terms of numbers of users and services, quantities of data stored and manipulated, rates of processing, numbers of nodes, geographical coverage, and sizes of networks and storage devices. Small scales may be just as important as large scales. Scalability means not just the ability to operate, but to operate efficiently and with adequate quality of service, over the given range of configurations. Increased capacity should be in proportion to the cost, and quality of service should be maintained.

The framework presented here, and described in a preliminary way in [1], has the following features which are lacking in previous work on scalability metrics:

- it separates the evaluation of throughput or quantity of work from quality of service,
- it allows any suitable expression for evaluating quality of service,

*This research was supported by the Natural Sciences and Engineering Research Council of Canada, through their program of Industrial Research Chairs, and by CITO (Communications and Information Technology Ontario.

- it adds to the system design a formal notion of a scaling strategy, which is a plan for scale-up. The plan can introduce different kinds of changes at different scales, since it often happens that all the components cannot be scaled simultaneously. This generalizes the notion of a scale factor, which becomes a parameter of the strategy,
- it incorporates scalability enablers, which express aspects of the design which should be tuned for efficient operation at any given scale.

Existing Scalability Analysis

A variety of scalability metrics have been developed for massively parallel computation, to evaluate the effectiveness of a given algorithm running on different sized platforms, and to compare the scalability of algorithms. These metrics assume that the program runs by itself, on a set of k processors with a given architecture, and that the completion time T measures the performance.

Three related kinds of metrics have been reported: speedup metrics, efficiency metrics, and scalability metrics. The following definitions give the flavour of the proposed metrics, although there are variations in detail among different authors:

- *Speedup* S measures how the rate of doing work increases with the number of processors k, compared to one processor, and has an "ideal" linear speedup value of $S(k) = k$.
- *Efficiency* E measures the work rate per processor (that is, $E(k) = S(k)/k$), and has an "ideal" value of unity.
- *Scalability* $\psi(k_1, k_2)$ from one scale k_1 to another scale k_2 is the ratio of the efficiency figures for the two cases, $\psi(k_1, k_2) = E(k_2)/E(k_1)$. It also has an ideal value of unity.

A typical metric is the *fixed size speedup*, in which the scaled-up base case has the same total computational work, and the speedup S is the ratio of the completion times (i.e., $S(k) = T(1)/T(k)$).

The above three metrics are described in [2], for a homogeneous distributed memory multiprocessor such as a hypercube or a mesh. The authors considered a fixed size case, a fixed time case and a memory-bounded case. In [3], a *generalized speedup* (taking a better account of memory access operations) is proposed for systems with shared virtual memory, and an *iso-speed scalability* metric for an algorithm-machine combination, which relates the workload capacity of the system at two different scales. In [4] an *isoefficiency* analysis is given, based on the question: at what rate should the problem size increase with respect to the number of processors in order to keep the "efficiency" fixed? In [5], a toolkit called the Modelling Kernel is described, which uses a model based on the program's parse tree. The choice of the scalability metric is left to the user. In [6], the techniques of experimentation, analytical modeling and simulation are compared, as they apply to studying parallel system performance. Scalability captures both the available and the delivered computing power, with the differences due to the overheads of parallel processing. The paper identifies overheads from different sources (hardware, software, algorithm), and summarizes metrics which include constant problem size scaling, time-constrained scaling, memory-constrained scaling and isoefficiency scaling.

The need for a new scalability metric and a methodology

Distributed systems require a more general form of scalability metric, because:

- Rather than running a single job to completion, these systems are shared by many jobs and new jobs arrive as others complete, so the behaviour should be modeled as a steady state.

- Throughput and delay should be evaluated separately as productivity factors. With a single job, the throughput is just the inverse of the job time; in a distributed system with an average of N jobs the throughput is N/(job time). The mean number of users adds a degree of freedom to the analysis.

- a greater variety of communications mechanisms may become involved, with their own scalability properties. Reliable multicast for example can introduce serious scalability problems.

- The productivity evaluation should be further expanded because there are more aspects to "adequate service" in distributed systems, called quality-of-service (QoS) figures. We shall use the term QoS here to include any measure of the goodness of a service (for instance, it could include a failure related or *availability* measure). For simplicity the examples considered in this paper are restricted to quality of service based on mean delay, but the framework includes any measure which can be evaluated.

- The "size" of the system is more complex because of the heterogeneous physical architecture of distributed systems. Instead of just a number of processors to measure size, one should consider symmetric multiprocessor nodes, replicated services, alternative networks and processors with different types and prices, and so forth. "Size" becomes multidimensional.

- Additional cost factors besides cost of processors, storage and bandwidth should be considered, such as the cost of software licenses, and perhaps the cost of operation such as management and help desks.

- The strategy for scaling up a distributed system is more complex than simply adding processors, storage and bandwidth. It may include replicating software services and storage, for instance, and modifying the communications mechanisms. An explicit scaling strategy is needed as part of the definition of the metric. This is a counterpart of the various kinds of parallel system scaleup defined in different metrics, (e.g. the fixed-time or fixed-speed scaleup metrics).

There has been a little previous work on distributed systems, in several distinct flavors. In [7], the scalability of Microsoft's Windows NT operating system is discussed using an in-memory subset of Microsoft's SQL server benchmark, to focus on the CPU performance. Scalability analysis is carried out by plotting a graph of performance figures obtained from this benchmark versus the number of processors. This study is quite close to the parallel systems studies, having homogeneous processor resources and ignoring software resources.

In [8], a scalable load monitoring service and a scalable resource query service for managing resources distributed across a network are described. In this work scalability means a linear relationship between the bandwidth requirements (i.e. the amount of traffic generated on the network) and the number of hosts on the network.

In [9], the authors argue that the existing solutions to provide transparent access to network services need to be modified for the internet paradigm, considering the scalability, fault tolerance and load balancing issues.

In [10], models are used to enable design-time modeling of complex large scale distributed applications. The authors analyze how some design parameters for the example system affect the application's QoS (defined as end-to-end mean response time) and scalability, with respect to the number of nodes and the number of domains.

In [11] a scalability metric suitable for distributed systems, called *P-scalability*, was examined. It employs the "power" measure $P(k)$ of Giessler [12], and the cost of all system resources at a scale

factor k, as follows:

$$\begin{aligned} \textit{P–scalability}\,(k_1, k_2) &= (P(k_2) \cdot \textit{Cost}\,(k_1))/(P(k_1) \cdot \textit{Cost}\,(k_2)) \\ P(k) &= \textit{(Throughput)/(Response Time)} \end{aligned}$$

This metric combines capacity and response time (both are present in the power P) with cost. However it has a defect (which is cured in [1] and the present work), in that it credits unbounded value to response times approaching zero. In fact however, for most users there is a required response time, below which further reduction has little or no value. This metric therefore distorts the scalability by rewarding very short responses which are actually not very useful.

In [1] there is a preliminary description of the metric presented here. The present paper adds a more complete description of the algorithms used to compute the metric, and of a number of idealized cases which validate its intuitive meaning. It introduces an upper bound on the metric, and gives the details of the scalability assessment of two substantial applications. It shows that the present metric is a generalization of some of the well-known metrics for scalability of parallel computations.

2 A Scalability Metric within a Scaling Strategy

The scalability framework is based on a *scaling strategy* for scaling up (or down) a given system, controlled by a *scale factor* k. We suppose that each scaled configuration is determined by a set of variables $(x(k), y(k))$ (which may take numeric values, or enumerated alternative choices), divided into two groups:

- $x(k)$ denotes a set of scaling variables, determined by the strategy for each value of k,
- $y(k)$ denotes a set of adjustable variables called *scaling enablers* which are tuned to maximize the productivity for any given k. Since k determines x by the strategy, and x influences y through the optimal tuning, the values of y are effectively determined by k.

Examples of scalability enablers are the allocation of processes to the processors, priorities, replication of processes and data, the creation of threads within processes, the memory available for buffers, tuning of the middleware parameters, network bandwidth and the choice of communication protocols. For a simple example, a database system might have a scaling strategy which defines the users, processors and the database size as functions of k:

- $\textit{Nusers} = k =$ the number of active users
- $\textit{Datasize} = 10000 \log_{10} k =$ the assumed size of the database, in records, as k increases.
- $\textit{Nproc} = \lceil k/100 \rceil =$ the number of processors to be provided, one per 100 users, rounded up.

Figure 1 illustrates this *scaling path* in the space (*Datasize, Nproc*), over the range $k = 100$ to 300.

At each value of the scale factor, the scaling strategy and the optimal values of the enablers determine the scaled configuration, from which the cost, capacity and quality values can be evaluated and used in the scalability metric.

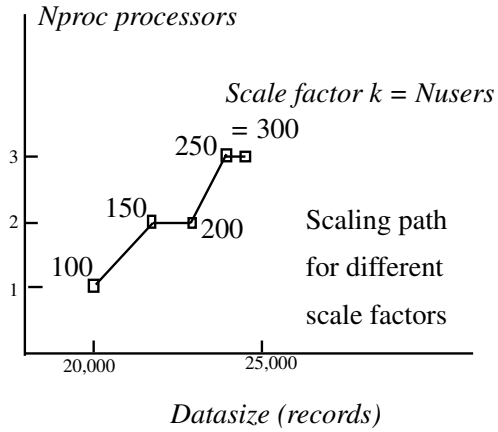


Figure 1: Scaling Variables and the Scaling Path

The Scalability Metric

The scalability metric is based on *productivity*. If productivity is maintained as the scale changes, the system is regarded as scalable. Given these three quantities:

- $\lambda(k)$ = throughput in responses/sec, at scale k
- $f(k)$ = average *value* of each response, calculated from its quality of service at scale k,
- $C(k)$ = cost at scale k, expressed as a running cost per second to be uniform with λ,

then the productivity $F(k)$ is the value delivered per second, divided by the cost per second:

$$F(k) = \lambda(k) \cdot f(k)/C(k)$$

The scalability metric relating systems at two different scale factors is then defined as the ratio of their productivity figures:

$$\psi(k_1, k_2) = (F(k_2))/(F(k_1)) \tag{1}$$

This is the scalability metric that is used in the rest of this paper. Frequently, k_1 is fixed at a known value and the metric is written as $\psi(k_2)$ or $\psi(k)$.

The system is regarded as "scalable" from configuration 1 to configuration 2 if productivity keeps pace with costs, in which case the metric ψ will have a value greater than or not much less than unity. In this work we arbitrarily use the threshold value of 0.8, and say the system is scalable if $0.8 < \psi$; the threshold value should reflect what is an acceptable cost-benefit ratio to the system operator. The value of k at the threshold is the scalability limit of the system. If ψ rises above 1.0 we will say the system has "positive scalability" (like superlinear speedup).

Of the three quantities that enter the metric, *throughput* is self-evident. The *cost* is not a one-time capital cost, but is expressed as a rental cost, to express costs and benefits consistently per unit time. Cost can include the cost of processor, storage, networks, software, management, help desks, etc. The present work will include only a few of these factors, for illustration. The *value function* $f(k)$ is determined by evaluating the performance of the scaled system, and may be a function of any appropriate system measure, including delay measures (mean, variance or jitter, probability of delay exceeding a threshold), availability, or the probability of data loss or timeouts. In this work,

for purposes of explanation and demonstration, we will consider only the mean response time $T(k)$ at scale factor k, compared to a target value $\hat{T}$, in the following value function:

$$f(k) = 1/(1 + (T(k)/\hat{T})) \tag{2}$$

With this value function, from (1) and (2) the scalability metric for scale k_2 relative to k_1 is, after a little simplification:

$$\psi(k_1, k_2) = \frac{\lambda_2 \cdot C_1 \cdot (T_1 + \hat{T})}{\lambda_1 \cdot C_2 \cdot (T_2 + \hat{T})} \tag{3}$$

Defining and Obtaining the Measures

In practice the choice of the function f will depend on the system goals and on what is practical to estimate. In the tradition of other scalability measures, the metric ψ is based on quantities that can be predicted by an analytic calculation. The calculation is more complex than in some other metrics, but it is carried out below using a well-established queueing or extended queueing analysis. If the scaled-up system is actually constructed and instrumented, the metric can also be calculated from measurements of its operation.

3 Scalability Metric Applied to Idealized Cases

To show the behaviour of the metric ψ, it is applied here to standard idealized systems which are widely understood at an intuitive level, which give analytic solutions for ψ. In all cases the value of k for the base case is taken as 1, and the metric is written as $\psi(k)$.

Case I: General speed-scaled open system with proportional costs

Here we consider *any* system architecture and behavior (not necessarily solvable analytically) which has external arrivals and a steady state. Configuration 1 is an arbitrarily chosen reference, and configuration 2 is uniformly sped-up or time-scaled by a factor k, so the input rate and every component (computers, networks, storage) is faster. Cost is scaled by a factor k so that $C_2 = k\,C_1$, and $\lambda_2 = k\,\lambda_1$, $T_2 = T_1/k$. A little manipulation of equation (3) gives:

$$\psi(k) = 1 + \frac{(k-1)T_1}{T_1 + k\hat{T}}, \quad \lim_{k\to\infty} \psi(k) = 1 + \frac{T_1}{\hat{T}} \tag{4}$$

This example has positive scalability with a bounded increase. If $T_1 \approx \hat{T}$ then ψ levels off around a value of 2.

This agrees with intuition. The throughput increase pays for the extra cost in exact proportion, and the shorter response time provides a bonus which is bounded because f is bounded as T goes to 0.

Case II: General speed-scaled closed system with fixed user population

In this case the system is closed instead of open, meaning that its load is generated by users or jobs which remain in the system, cycling and creating a new request as soon as the previous response is over. When there are N users or jobs and a mean response time T, the system throughput is $\lambda = N/T$. Again we consider any system architecture and behaviour with a single class of users, that is all jobs make the same demands, and an arbitrary starting point with $k = 1$. The cost is scaled by k, but the number of users is fixed at N and is not scaled, and the target mean response time is also fixed.

Configuration 2 is again uniformly sped-up or time-scaled by a factor k, so every device and server is faster. Thus just as in Case I, $C_2 = k\,C_1$, $\lambda_2 = k\lambda_1$, and $T_2 = T_1/k$, and $\psi(k)$ is again given by Eq. (4).

The intuition about this system is identical to that in Case I and the results agree in a similar way.

Case III: Closed balanced ideal queueing network, scaled in users and speed.

This case is like the last one but more restricted in its assumptions about the system, which has a separable queueing network performance model with N jobs and k servers, with equal demands D sec. per response to all servers, and single, constant-rate servers. The throughput and response time are [13]:

$$\lambda = \frac{n}{[(N+k-1)D]}, \quad T = \frac{n}{\lambda} = (N+K-1)D$$

In Configuration 2, the number of jobs N, and the server speeds and costs are scaled by a factor k. Thus Configuration 2 has server demands of D/k sec per response, at each of the K servers. Substituting into equation (3) with $N_2 = kN_1$, $D_2 = D_1/k$ and $C_2 = kC_1$ we get

$$\psi(k) = \frac{k(N_1+K-1)}{(kN_1+K-1)} \cdot \frac{(N_1+K-1)D_1 + \hat{T}}{(kN_1+K-1)\left(\frac{D_1}{k}\right) + \hat{T}} \tag{5}$$

$$\lim_{k\to\infty} \psi(k) = \frac{(N_1+K-1)[(N_1+K-1)D_1 + \hat{T}]}{N_1[N_1 D_1 + \hat{T}]}$$

By inspection this limit is greater than 1.0, but if $K \ll N_1$ the limit approaches 1.0.

Case IV Asymptotic general closed system scaled in users and speed

This combines the two previous cases, but only considers the asymptotic condition in which the system is effectively bottlenecked at its slowest server, even in the base configuration. The system throughput is effectively determined by the demand at this server, giving $\lambda \approx 1/D_{\max}$. In configuration 2, $D_{\max,2} = D_{\max,1}/k$, and thus throughput follows $\lambda_2 \approx 1/D_{\max,2} \approx k \cdot \lambda_1$. As $N_2 = kN_1$, and $C_2 = kC_1$, the response time is roughly constant at $T = kN/k\lambda = N/\lambda$, and $\psi \approx 1.0$.

Case V: System with a single non-scalable bottleneck and increasing users

Now consider the same case of a general closed system with a bottleneck, as in Case III, in which the bottleneck device cannot be speeded up, which in turn limits λ to some constant value $\lambda_{\max}$. The population N and costs C are proportional to k. Then for large k $T \to N/\lambda_{\max}$ and:

$$\psi(k) \approx \frac{\lambda_{\max}}{\lambda_1} \cdot \frac{C_1}{kC_1} \cdot \frac{N_1/\lambda_1 + \hat{T}}{kN_1/\lambda_{\max} + \hat{T}} \to \left(\frac{\lambda_{\max}}{\lambda_1}\right)^2 \frac{1}{k^2}. \tag{6}$$

Thus the scalability declines as $1/k^2$, for *any* closed system that has a single dominant bottleneck.

Case VI: Closed balanced system scaled by replication of servers, and user population

As in Case III, this system has a separable queueing model with balanced demands on K servers. To scale it up, each server is replicated k times and its load is divided equally among the k replicas of each

server. If there is no overhead for managing replicas, it seems intuitively clear that this is a perfectly scalable system. Consider the scaling path:

$$N_2 = kN_1; \quad K_2 = kK_1; \quad D_2 = D_1/k; \quad C_2 = kC_1$$

Following Case III, but with $K_2 = kK_1$ we find that $\psi > 1$ for all k, and for large k the limit is almost the same:

$$\lim_{k\to\infty} \psi(k) = \frac{(N_1 + K - 1)[(N_1 + K - 1)D_1 + \hat{T}]}{N_1[(N_1 + K)D_1 + \hat{T}]}$$

Again if $K \ll N_1$, $\psi \to 1.0$. For an unbalanced system the result is also the same.

Case VII: Effect of overhead costs

In Case VI, suppose that for $k > 1$ the demands D are augmented by coordination overhead, for example to maintain overall system state data. The replicas, costs and user population are all scaled by a factor k. An efficient coordination mechanism might limit the overhead cost to a slowly increasing amount of overhead, giving a scaling relationship for the server demands such as

$$D_1 = (D_1/k) + D_0 \log k$$

Then, once again following the approach of Case III, Eq. (3) gives:

$$\psi(k) = \frac{k(N_1 + K_1 - 1)D_1}{(kN_1 + K_1 - 1)(D_1/k + D_0 \log k)} \cdot \frac{(N_1 + K_1 - 1)D_1 + \hat{T}}{[(kN_1 + K_1 - 1)(D_1/k + D_0 \log k) + \hat{T}]} \tag{7}$$

which decreases slowly towards zero as $k \to \infty$. Thus, scalability is moderate for "small" values of k.

If the baseline system is quite large, so that N_1 is large compared to K_1 and to $\hat{T}/D_1$, then (7) can be simplified to $\Psi \approx k/(1 + (D_0/D_1)k \log k)^2$, which is greater than unity for values of k that satisfy the approximate inequality $\sqrt{k} \cdot \log k < D_0/D_1$. As an example, suppose that D_0/D_1, which describes the relative magnitude of the coordination overhead, is 0.1. Then at $k = 10$, this gives $\psi \approx 2.5$, and $\psi > 1$ for values of k up to nearly 40. A larger overhead ratio will limit the scalability more.

Case VIII: A closed system with scaled population and target response times

This case considers a scaling path in which response degradation is *accepted* in proportion to a rising user population in an otherwise *unscaled* system. This is quite a different system goal, and illustrates the flexibility of the framework. We consider the balanced closed system of Case III, but with constant values of C and D. The analysis of Case III then gives the value function

$$f = 1/(1 + T/\hat{T}) = 1/(1 + (N + K - 1)D/\hat{T})$$

If $\hat{T}_2 = k\,\hat{T}_1$ and $N_2 = k\,N_1$, then $\frac{d}{dk} f > 0$, and it is also well-known that $\frac{d}{dk} \lambda \geq 0$. Then it can be deduced from Eq (3) that ψ is an increasing function of k, and that it approaches a constant limit greater than 1. This is a symptom of the well-known fact that the response time rises at the same asymptotic rate as the population.

The conclusion is that, if response time degradation is accepted in proportion to users and is included in the scalability function, closed balanced systems are infinitely scalable in population.

Case IX: A single scaled open multiserver queue

Often one tries to scale up a system by adding servers. This case considers an ideal multiserver queue ($M/M/m$ queue) in which: the number of servers m, the arrival rate λ, and the cost C are all scaled by a factor k. There is no server coordination overhead, and the queue shares the load in an ideal fashion, so the metric should show infinite scalability.

The solution is well-known but lengthy, so it will not be shown here, but it does show that $\psi > 1$ for all k, which supports the intuition.

As $k \to \infty$, the metric approaches the form:

$$\psi \to \left(1 + \frac{1}{\hat{T}}\left(S + \frac{\rho \cdot P_Q}{\lambda \cdot (1-\rho)}\right)\right) \Big/ \left(1 + \frac{S}{\hat{T}}\right) > 1 \tag{8}$$

where S is the service time at any server, λ is the arrival rate, ρ is the utilization of each server ($\rho = \lambda S/m$) and P_Q represents the Erlang–C formula for the probability that all m servers are busy. The fraction with $\rho \cdot P_Q$ in the numerator approaches zero with large k, and ψ approaches 1 in the limit.

Summary

The cases I–IX cover a wide range of well-understood systems and of scaling policies, and reveal how the metric proposed in this paper will evaluate different kinds of systems, and agrees with intuitive judgement. This gives some confidence in applying it.

The parallel-system metrics surveyed in Section 2 also fit into the general framework of this paper as special cases. If we consider a steady state with one job at a time being executed, one after another, and fixed-size scalability, then a parallel computer is a closed system with replication and overhead, as in Case VII (except the user population is not scaled). The time to completion is rewarded through the throughput term, with $\lambda = 1/T$. The QoS function considered in most metrics is simply $f = 1$, since they only evaluate the time to completion and that is already taken into account in the throughput.

4 Procedure for Scalability Analysis

The analysis in the previous section depended on closed form solutions which are usually not available Table 2 shows a procedure which uses practical numerical methods:

- numerical performance approximations to calculate the productivity measure for each scaled system, given the values for the sets x and y of system parameters. In this work, response time and throughput were calculated.
- numerical search techniques to maximize the productivity measure over the scaling enabler variables y. The performance model is solved at each step of the search.

The procedure of Figure 2 will calculate ψ for the scale-up from a given reference system to a scaled version for some value of k including the possibility of comparing multiple candidate scaling paths.

The performance evaluation can be done with any suitable model. The models used in this work are analytic layered queueing networks or LQNs ([14], [15]) which are a kind of systematic extended queueing model that use well-known approximations for solution. LQNs were designed to evaluate distributed systems. They directly represent software servers as servers with queues, and several kinds of software resources which must be scaled, including mutexes and process pools or thread pools.

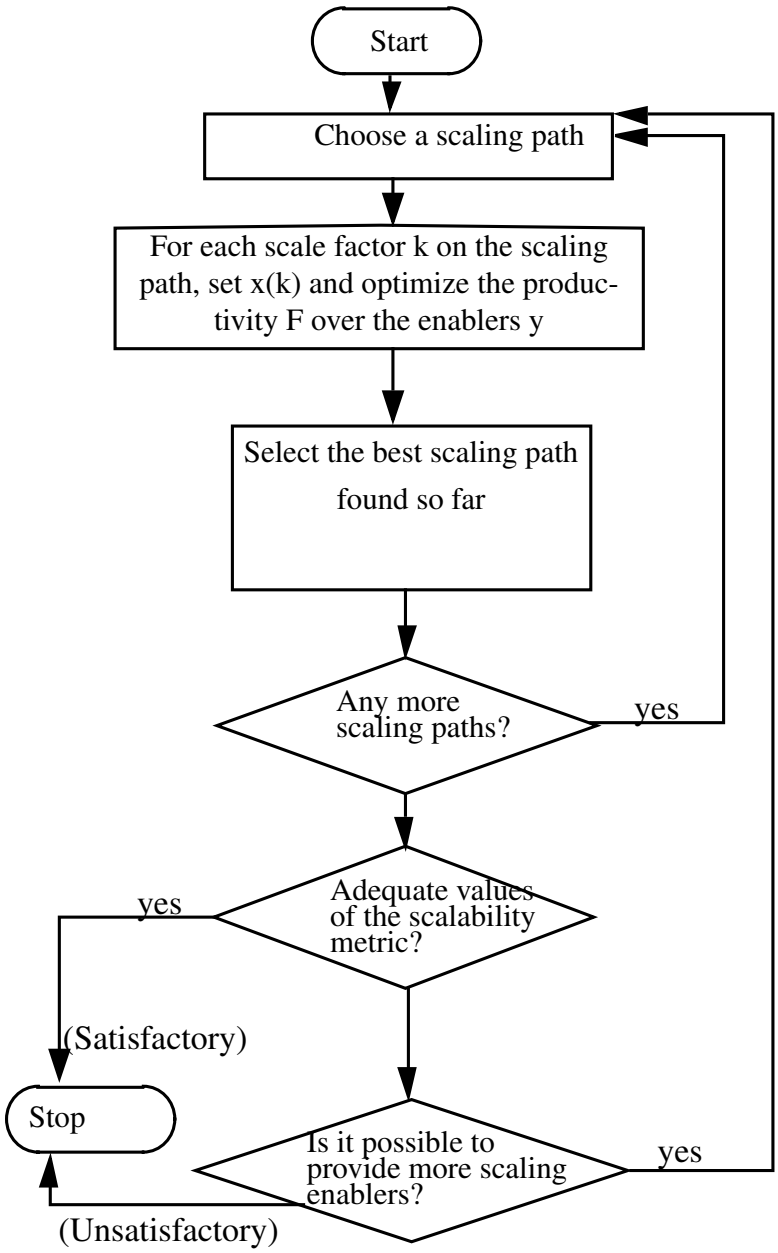


Figure 2: The algorithm for evaluating scalability

Optimizing the productivity metric using simulated annealing

In order to maximize the productivity of a particular configuration by tuning the scalability enablers y, this research used the simulated annealing algorithm described in ([16], [17], [18]). It was used because it is robust, in the sense that it can handle a wide variety of relationships, including discontinuous functions and integer or categorical variables. Its disadvantages are that it can consume very many search steps and it gives no guarantees about convergence.

Simulated annealing takes random steps controlled by a parameter called the "temperature" τ. The productivity function is evaluated and the perturbation is accepted if it gives an increase, or is either accepted or rejected if it gives a decrease. The acceptance probability is smaller for a greater decrease, and as τ decreases this probability also is reduced. Termination was decided if the fraction of the accepted moves was less than 2% for two successive full iterations, or if the number of iterations exceeded a predefined limit. The best solution found was retained and used as the final result.

5 An optimistic bound on the scalability metric

Optimistic assumptions about resources can greatly simplify the calculations, and at the same time give a bounding value on productivity. First, the optimistic assumption gives an upper bound on performance, and thus on the value function. Then further assumptions may be able to do away with the need for optimization, and allow the use of a single evaluation at each scale factor.

The bounds described here are only for quality measures based on the mean response times. By ignoring the constraints imposed by some resources (such as locks and critical sections), and making optimistic assumptions about overhead costs and task execution demands, an analytic queueing model results (although not usually as simple one as the cases described in Section 3). The resulting value of ψ should always be larger than the true value, and may be quite a bit larger. However, the bound does capture the effect of the raw balance of power and demand along the scaling path, as represented by the total demands for execution operations on the set of devices. And if the bound shows scalability is inadequate, the more detailed calculation will show it even lower. In the major example in the next section the bound gave a useful indication of the more detailed result.

Client-server systems are usually "closed", in the sense that they contain a certain number of users who issue requests into the system, and wait for responses. If one ignores software resources (such as process threads and memory), and makes a few other assumptions, they may be modelled as closed queueing networks. Then two of the "balanced job bounds" in [13] give an upper bound on the throughput, and a lower bound on the mean response time. These bounds evaluate the system with its total workload spread equally across all the processors and devices. This at least correctly captures the increase in processing power, the replication of services in the scaling strategy, and the overheads associated with the scaling path (e.g. consistency management overheads associated with replicating a database).

Because the productivity is an increasing function of throughput (which is overestimated by the bound) and a decreasing function of response time (which is underestimated), the productivity and scalability calculated using the bounds are always overestimated. This supposes that the exact cost factors can be used, and that the base case is correctly evaluated.

Algorithm for the scalability bound:

The steps in calculating the scalability bound are then:

Step 1: Determine the productivity for the base case, $F(1)$, by a detailed calculation.

Step 2. For each scale factor k, determine the scaled system configuration from the scaling strategy. Compute the total seconds of execution of each device, averaged per response, as follows:

- Execution and overhead which is determined and assigned to each device by the scaling strategy is calculated first,
- The remaining execution demand is added up over the remaining tasks and spread (optimistically) over all the devices so as to produce the most even distribution of the total demand, expressed in seconds of execution per response. That is, it is allocated without regard to allocating entire tasks to one device, but with regard to whether the device can do the work (so, CPU demand is spread over CPUs and disk demand over disks).

Optimistic assumptions about overheads mean that they are set to the lowest value consistent with the scaling strategy; thus if two tasks included in the remaining demand should (by the scaling strategy) be allocated separately, internode communications overhead is included.

The result of this step is a set of demands which may still be unequally distributed over the devices, because of constraints in spreading the workload.

Step 3. At scale k, set $C(k)$ to the cost of the scaled system and, following ([13] chapter 5), find bounds on λ and T:

- set $\lambda(k)$ to the *minimum* of (1) the balanced system throughput bound for a queueing network with the same servers, and (2) the asymptotic throughput bound for the given set of demands
- set $T(k)$ to the balanced job value
- compute $F(k)$ from Eq. (3).

Step 4. Set the scalability metric bound to $\psi = F(k)/F(1)$, and then the bound-based scalability limit is the first value of k giving a y that drops below the "moderate scalability" limit of $1 - \varepsilon$.

The queueing network model with the evenly spread workload is constructed so that it intuitively gives a performance bound, however the relationship is not rigorously proven. The intuitive reasons for believing it gives a bound are

- software resource constraints are ignored, which can only improve performance,
- allocation decisions which are enablers in the strategy are represented in the bound by the greatest possible degree of load balancing, which should give better performance than the best feasible allocation that respects task granularity,
- overhead that is not explicitly required by the scaling strategy is omitted.

The bounds can show the consequences of changing demands and power with k. Suppose that the scaling strategy resulted in a total demand (in seconds of execution, adding over all nodes) of $D(k) = g_1(k)$, the number of nodes (all equally fast) is $g_2(k)$, and there is a user delay (not included in the response time) of Z_0. Then the bound calculation is:

$$\begin{aligned} D_{\text{avg}}(k) &= D/g_2(k) = g_1(k)/g_2(k) \\ R(k) &= D(k) + (N-1)D_{\text{avg}})/(1 + Z_0/D(k)) \\ &= g_1(k) + (N-1)(g_1(k)/g_2(k))(g_1(k)/(Z_0 + g_1(k))) \\ T(k) &= R(k) + Z_0 \end{aligned}$$

The bound on the scalability metric can then be expressed as:

$$\psi_{\mathrm{bnd}}(k) = \frac{F_{\mathrm{bnd}}(k)}{F(1)} = \frac{\min\left\{\frac{kN}{Z_0+g_1(k)+(N-1)\frac{g_1(k)}{g_2(k)}\cdot\frac{g_1(k)}{(Z_0+g_1(k))}}, \frac{1}{D_{\max}}\right\}}{C(k)\Big(1+\frac{1}{T}\Big(Z_0+g_1(k)+(N-1)\frac{g_1(k)}{g_2(k)}\cdot\frac{g_1(k)}{(Z_0+g_1(k))}\Big)\Big)\cdot F(1)} \tag{9}$$

When the system is saturated, both the numerator and denominator are dominated by the terms in the big round brackets multiplied by $(N-1)$. The direct effect of adding work (increasing $g_1(k)$) is always to decrease ψ. The direct effect of adding nodes is to increase $g_2(k)$ and $C(k)$ both, so as far as the bound is concerned the effect is neutral when the system is saturated, and harmful to scalability when it is not. The direct effect of causing a bottleneck node, due to a scaling path that does not allow the load to be properly balanced, is to increase $D_{\max}$ and decrease scalability through the last term in the numerator. All of these effects are expected, but the equation gives a picture of the order of the relationship.

A second version of the bounds analysis, which is closer to a kind of approximation, is to use the bounding value for performance and productivity in the base case also. This puts all scale factors on an equal footing as regards the looseness of the bounds, however it reduces the certainty that the value of ψ_{bnd} is in fact a bound since the denominator may be overestimated.

6 A connection-management system

This section analyzes the scalability of a connection management system, based on the design and parameters of a real industrial prototype. It is a design which evolved out of a connection-management design described previously in [1] and [11].

Figure 3 shows the major components in a prototype connection management system for virtual private networks, intended to support applications such as video-conferencing. The prototype was heavily influenced by standards such as G.805 [20]. It was designed to be able to:

- set up a virtual private network joining user-specified end-points, and allocating the network resources in such a manner as to meet the QoS requirement,
- manage a variety of heterogeneous switching equipment, for the purpose of setting up end-to-end connections,
- use the allocated resources of the virtual private network and let the user set-up/tear down connections arbitrarily, among any of the sites.

The prototype was implemented using a network of workstations running UNIX, with DCE middleware to handle intertask communications and transparency, and a backbone network based on a SONET OC-12 (622 Mbit/s) optical fiber ring with proprietary switching equipment on which cross-connections can be made or released as required. The software tasks can be roughly classified into three logical layers:

- The *topology layer*, that deals with the connection topology of the virtual private network (VPN), connecting all the user-specified endpoints (e.g., the User-Network Interface identifiers – UNI's – in the case of an ATM network). Once a virtual private network is established, the objects in the topology layer can directly communicate with the lowest layer (called SONET here), in order to set up virtual channels over this VPN.

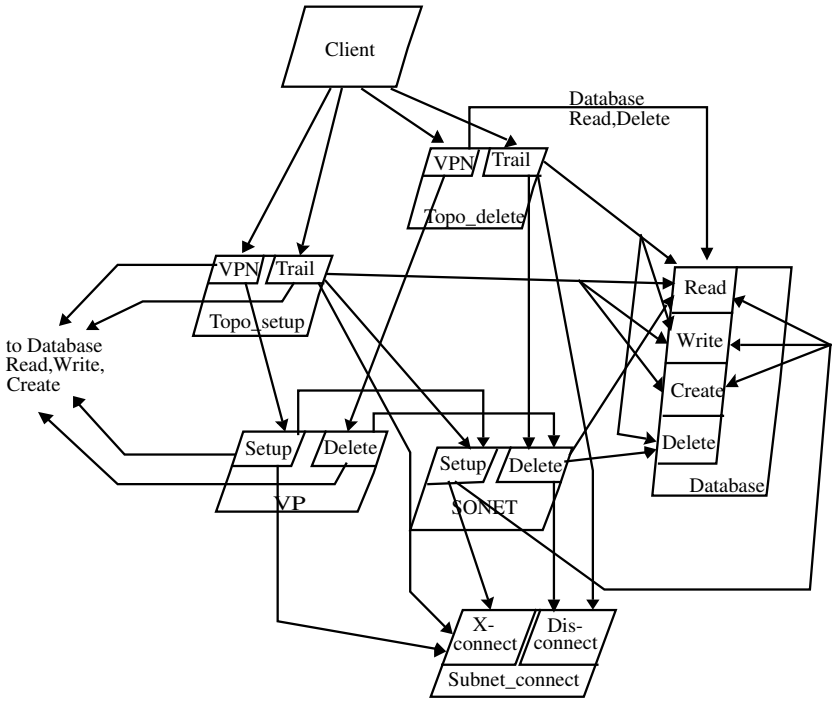


Figure 3: The Connection Management System Prototype

- The *virtual path* (VP) layer, that deals with connecting all the sites in a virtual private network with a virtual path. This corresponds to provisioning the network resources to meet user-specified bandwidth and QoS, to support future connections.
- The *SONET* layer that supports a virtual path by setting up appropriate connections on the SONET ring.

  Following is a brief description of the tasks in Figure 3:

  - The *client tasks* represents the users that set up (or dismantle) the virtual private network and set up (or dismantle) connections on an existing virtual private network. The clients could be the software tasks that manage higher level applications, e.g. a video conferencing system that uses the given connection management system.

  The clients interact with the topology layer to set up the virtual private network as well as the connections on it (VC's or the virtual channels). The frequency of setting up/releasing a VPN, which is like a leased line, is much lower than that of setting up/releasing temporary connections, by a ratio of 1:50.

  - *Topo_setup* and *Topo_delete*: these tasks belong to the topology layer discussed above, and support setting up VPNs as well as connections within a VPN. The necessary routing functions are built into the setup entries of these tasks and of their servers.
  - *VP*: This task sets up and deletes virtual paths (VPs) that make up a VPN.
  - *SONET*: This task manages the fibre-level port-to-port connections required to support the setting up of the VP layer trails, which in turn help set up the VPN.

- *Subnet_connect*: This task directly controls the SONET network elements.
- *Database*: The database stores objects related to the various functional layers in the system, and provides state data to all the functions.

The database, which is accessed heavily by almost all the tasks in the system, clearly is a potential hot spot in the system. By measurement it was verified that the database indeed had the greatest demands for both VPN setup/release as well as connection setup/release, and would limit scalability if its capacity were not increased. One approach to this is database replication, which was considered as an element in the scaling strategy. As we shall see, the hazard in replication is heavy overhead.

The prototype system was instrumented and measured to obtain workload parameters for the performance model which was used to evaluate the scalability.

Scaling strategy for the connection management system

The scaling strategy was to introduce replications of the database, using the location-based replication paradigm described by Trantafiliou and Taylor in [21]. For each database replica, an additional processor was also added to the system. (We note that the location based paradigm was motivated by reliability as well as performance, and the reliability effects are not rewarded in the value function f used here.)

The scale factor was set to be the number of database replicas. A fixed number of five proces- sors was provided to run the other tasks in a fixed configuration, and the number of users was taken as a scalability enabler. Further enablers that were not used could have been the allocation of the tasks other than the database tasks to the processors, and replicas and additional processors for the other functions.

For each scale factor a performance model was set up with the replicas and their overheads, with overhead amounts calculated from the number of replicas, and the requests sent from any client entry to the database task were equally divided among all the replicas. The fixed remote invocation overheads were incorporated in the execution demands of the task entries. The fact that the accesses to the database replicas were symmetric happens to permit a special efficient approximation for symmetric replication of subsystems to be used in the solver [22].

In order to model the consistency management overhead (in terms of extra execution), each replica of the database is associated with a transaction overhead pseudo-task on the same CPU. The transaction overhead task accounts for the synchronous and asynchronous broadcasting overheads, locking overheads, etc. for consistency management, and the calls made by the database entries to the overhead task during the operation, prepare, commit and abort phase are proportional to the number of database replicas in the system.

The number of write transactions is significant, but the granularity of the database objects is small, so the probability of conflict on locks was assumed to be negligible and lock queueing delays were not modelled. However, the execution overheads of locking were substantial and were included.

The response of the system was modelled as a cycle of effort for one conference, including setting up and tearing down 5 virtual channels for a video conference between the two sites, plus one time in ten it included setting up a VPN as well. The cycle had a target time of 15 minutes ($\hat{T}$ = 15 min.). Load was generated by a number of users, who were modelled as having a "thinking time" of 10 minutes, between one cycle and the next.

The provisioning cost for the base configuration, including one copy of the database server, and one processor per software task, is taken as \$100,000. Each extra copy of the database server (including a new dedicated processor) is assumed to cost an additional \$5000. This gives a cost per unit time of the form $Constant^*(1 + 0.05k)$.

The reference configuration of the system had a single database copy, and was also optimized with respect to the number of clients, giving a reference productivity of 702 cycles of activity per hour per unit cost, and a reference throughput of 95 cycles of activity per hour. (That is, setting up and tearing down 9.5 virtual private networks, and setting up and tearing down about 475 virtual channels per hour).

6.1 Scalability bound

Step 1. The base configuration with 6 processors is optimized with respect to the number of clients, to obtain 23 clients, 95 operation units per hour and productivity $F = 1.95 \times 10^{-5}$ units/hour.

Step 2. At each scale factor, with k database replicas and k database processors, the balanced demand is calculated, including the overheads. In this case,

$$
\begin{aligned}
&\text{total demand, } D = 14.44 + (22.11k) \text{ sec.,}\\
&\text{average demand} = D_{\text{avg}}(k) = (14.44 + (22.11k))/(k+5) \text{ sec.,}\\
&D_{\max} = 35.08 \text{ sec.,}\\
&\text{response time} = D + (N-1)D_{\text{avg}}/(1 + (Z/D))\\
&Z_0 = 600\text{sec.,}\\
&\text{cost } = C(k) = 1 + 0.05k \text{ units/sec.;}
\end{aligned}
$$

Steps 3, 4. The solution gives the response time $T = D + (N-1)D_{\text{avg}}/(1+(Z/D))$ and throughput $\lambda = N(Z_0 + T)$ for the balanced system. Substituting into Eq. (9), we get the following expression for the scalability bound:

$$
\psi_{\text{bnd}}(k) = \frac{5.13 \times 10^4 \min\left\{ \frac{kN}{Z_0+D+(N-1)\frac{D}{(k+5)}\cdot\frac{D}{(Z_0+D)}} \,,\ \frac{1}{D_{\max}} \right\}}{(1+0.05k)\Big(1 + \frac{1}{T}\cdot(Z_0 + D + (N-1)\frac{D}{(k+5)}\cdot\frac{D}{(Z_0+D)})\Big)}. \tag{10}
$$

The total demand, the number of processors, and the cost all grow linearly with k. For large k, the scalability metric bound drops as k^{-2}. In fact it is the increasing overhead demands which cause the devices to saturate, and limit the scalability. The equation gives the plot in Figure 4. If the acceptable scalability limit is 0.8, it is reached at scale factor of $k = 8$. This is similar to the conclusion obtained in the next section, which derives a limit of 5 from a more detailed analysis.

6.2 Scalability metric: full calculation

The full calculation optimizes the productivity function with respect to the available scalability enabler (the number of clients), at each scale factor. The results are summarized in the following table.

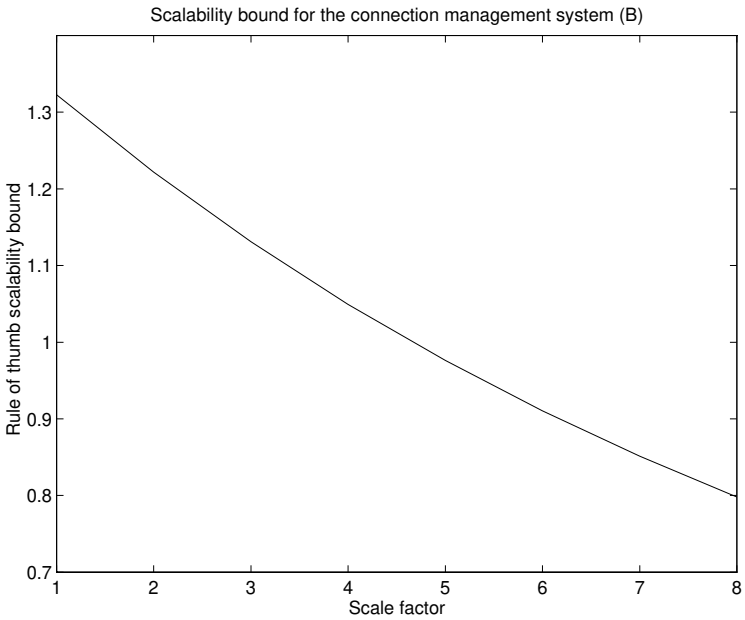


Figure 4: Rule-of-thumb scalability bound for the connection management system

TABLE 1. Scalability analysis results for the connection management system.

Scale Factor	Productivity (Optimized) (sec^{-1}per unit cost)$\times 10^{-2}$	Scalability metric value (Optimized)	Throughput (operations per hour)	Normalized* Response Time	Database CPU Utilization		System Cost (units)
					Total	Due to transaction overheads	
1	1.95485	1.0	95	0.3017	92.59	–	1.05
2	2.02031	1.0335	108.62	0.3645	86.86	67.85	1.1
3	1.90128	0.9726	111.18	0.4126	82.43	69.45	1.15
4	1.75662	0.8986	112.01	0.4761	79.77	69.97	1.2
5	1.61546	0.8264	112.26	0.5449	77.98	70.12	1.25
6	1.4861	0.7602	110.95	0.5953	75.77	69.30	1.3
7	1.36948	0.7006	110.94	0.6675	74.84	69.30	1.35
*The normalized response time is the mean response-time divided by the target of 15 min.							

The Table shows that the scaling strategy and optimization give response times which are well within the target at all scales, but scalability is only moderate. The throughput increases from $k = 1$ to $k = 2$ and then levels off, while costs rise, which drives the scalability down. The Database CPU columns show that most of the database work is overhead, at the larger scales. Figure 5 shows the detailed scalability measure and the bound plotted together.

The results show the system is spinning its wheels, generating overhead but not performance. The rate of setting up and deleting video conference connections is increased from 475 to 555/hour. The database costs rise to about 30% of the initial cost which included just one database.

Even though it reaches a scale factor of 5, the useful throughput increases by less than 20%. This emphasizes the fact that the scale factor defined in this work is just an index into the plan; it isn't itself a measure of the increase in productive work. The replications are cheap but don't achieve much

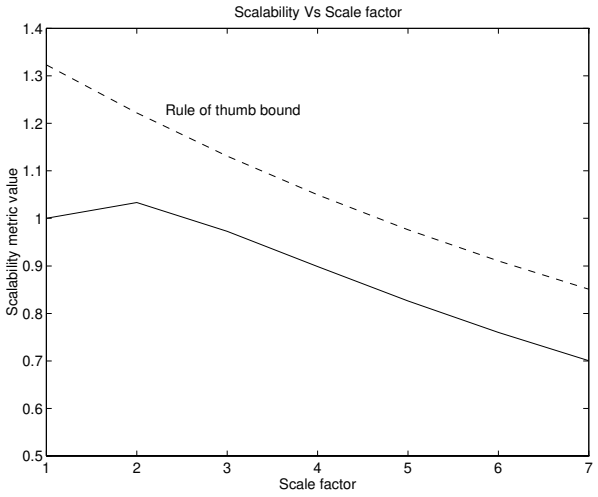


Figure 5: Scalability by the detailed calculation and by the bound: Connection Management System

for productivity. The overall ratio of the read to write operations is approximately 2:1. A higher read-to-write ratio would give less coordination overhead, and greater scalability.

For further scale-up this system, the results indicate some possible directions:

(1) The database schema could be re-worked to reduce the number of separate transactions.

(2) The routing algorithms at the topology, VP and SONET layer made heavy use of database transactions, and could be redesigned to reduce the database operations.

(3) It might be possible to *partition* the database objects, instead of replicating the whole database. For example, in this case, the ATM objects and the SONET objects could possibly be partitioned into two disjoint parts, by redesigning the database schema.

7 Scalability Analysis of a Call Processing System Prototype

The second example is a prototype call-processing system for digital telephony, based on proprietary message-oriented middleware. The objective of the evaluation is to assess:

- up to what point a product would be scalable, if built using the same basic design decisions,
- how investment should be made in the hardware and software components of the system for supporting different numbers of users,
- the impact of the location-service based replication model for database transactions, [21].

7.1 The architecture of the call processing system:

This system differs from the traditional call-processing systems used in digital telephony. It is implemented using a message-oriented middleware based on a fast ATM network. The objective is to minimize the latency, and to bypass the overheads in RPC stubs, the TCP/IP protocol, etc. in order to make the distributed call processing as fast as possible.

The system is based on some of the concepts described in [23]. The U-Net communication architecture uses a virtual view of the network interface that can be directly accessed at user level,

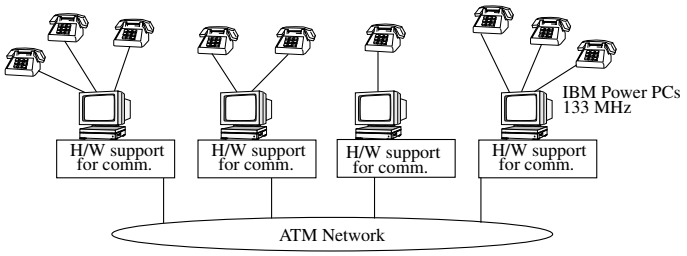


Figure 6: Overview of the call-processing system's hardware infrastructure

allowing direct access to high-speed communication devices. Removing/bypassing the communication subsystem's boundary with the application-specific protocols achieves efficient communication protocols with reduced system call overheads, and more importantly, allows buffer management at the user level. Multiplexing/de-multiplexing is embedded directly into the network interface. Thus, the network interface is virtualized, and each process has an illusion that it owns the direct interface to the network. This enables abstracting out the network interface for some applications, while still supporting legacy protocols through the kernel.

Dedicated ATM virtual channel circuits are used for call setup, with additional channels for the actual voice traffic. This scalability analysis assumes that there is sufficient ATM network capacity for the voice traffic, so it is not modeled. The model concentrates on the objects that collaborate to set up an end-to-end call connection.

The processing steps and the hardware platform

Figure 6 shows the hardware platform used in the prototype call-processing system. The hosts (IBM Power PCs running AIX, 133 MHz) are connected to an ATM network. Each host has the call processing software that supports the two half-call model, one half for call origination and one for call termination [24].

Any process that wishes to access the network creates one or more objects called endpoints, allocates memory for storing the messages (called the *communication segment*) and creates a set of send, receive and free message queues with each endpoint.

To send a message, a user process composes the data in its communication segment and pushes a descriptor for the message onto its send queue. The network interface, which is embedded in an integrated device driver, then picks up the message and sends it over the existing connection in the ATM network. Incoming messages are demultiplexed and transferred to the appropriate communication segment, and a message descriptor is pushed on the corresponding receive queue. Each process polls its receive queue periodically, to check for arrivals.

The system has a "true zero copy" architecture in which the data is transferred from the sender's communications segment, to the receiver's, without intermediate buffering. The communication segments span the process address space and the sender specifies an offset within the destination communication segment, at which the message data is to be deposited directly by the network interface.

Error checking and correction is handled at the application level. In this work it is assumed that the communications medium is sufficiently reliable, and sufficient memory is available, to let us ignore the performance effects of recovery from errors and buffer overflows.

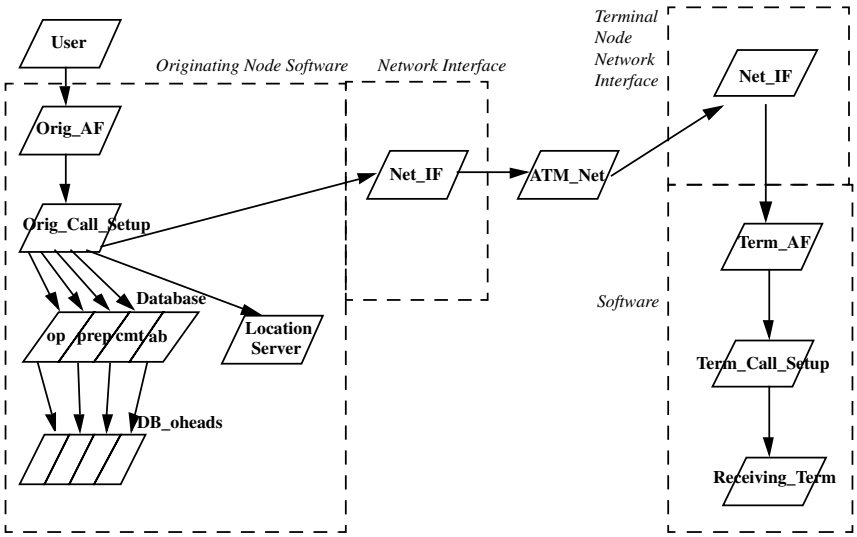


Figure 7: Layered Model fo the Call Processing Software, including the processes and Interactions

Software organization and layered model

Figure 7 shows the software involved in a call, at the level of processes and interactions. It actually shows a layered queueing network model which, as well as the software tasks, includes components for all the Users, the hardware network drivers, and the network delay, as "tasks" with their own delays. However it does not model the individual software objects. The prototype which was measured had two nodes, and invoked a total of about 80 objects per call, among all the collaborating processes.

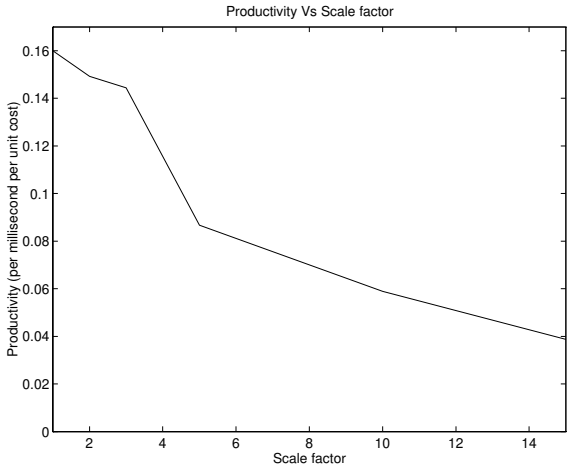
Call setup is managed by an originating half-call-setup process communicating with a terminal half-call-setup process. The originating process uses a location server to determine which database replica to use, and then obtains authentication data and network addresses from the database. Almost all the database requests in normal operation are reads, since writes only occur for new customers, changes in a customer's service profile and changes in the physical network.

Each node has a sufficient number of agents, and may or may not have one or both of a database replica and a location server replica.

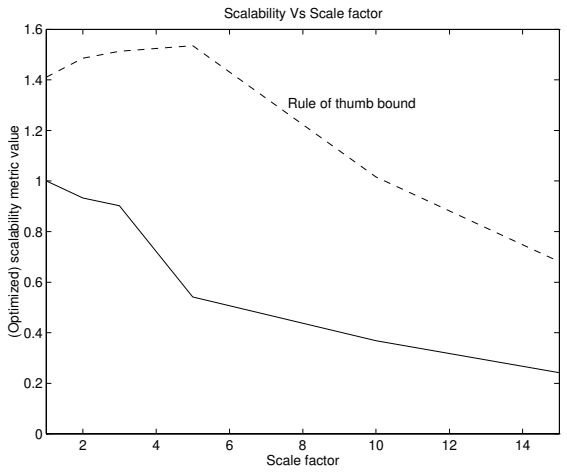
Scaling strategy and results

The number of CPUs in the system was chosen as the scale factor, and the scalability enablers are the allocation of software objects, replication of the database and the location server, and the number of clients. Threads are not allocated explicitly, since they are assumed to be inexpensive, and are provided as a thread-per request.

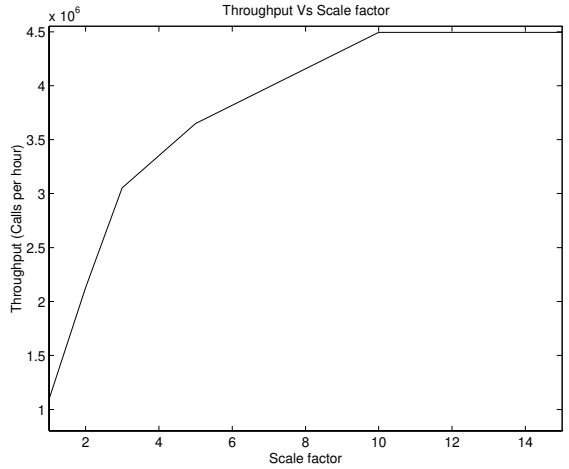
The scalability was analyzed over a scale factor interval of 1-15. The results of the optimization are presented below, in Table 2. The same data are plotted in Figure 8, along with the results of the bounds analysis.



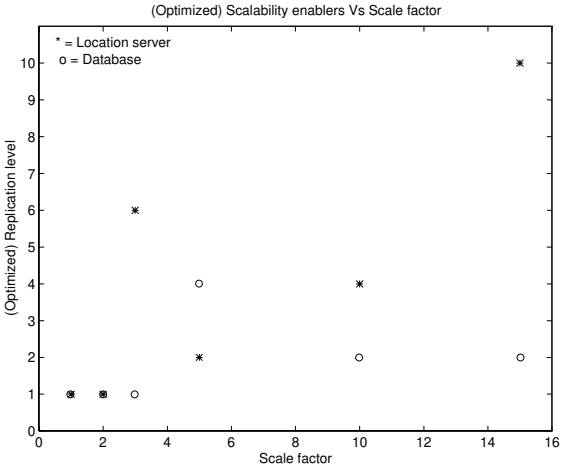
(a) Productivity VS the scale factor



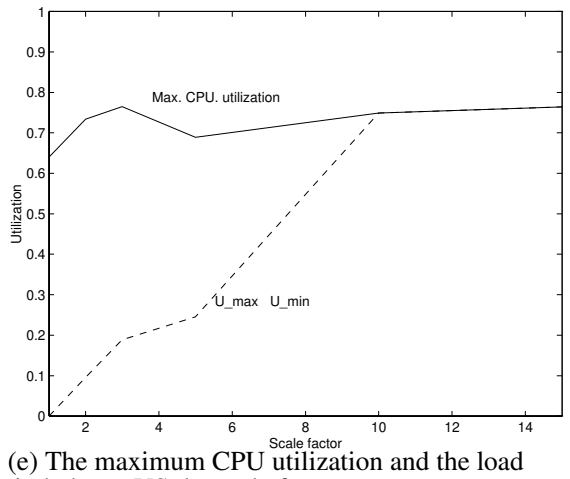
(b) Scalability VS the scale factor



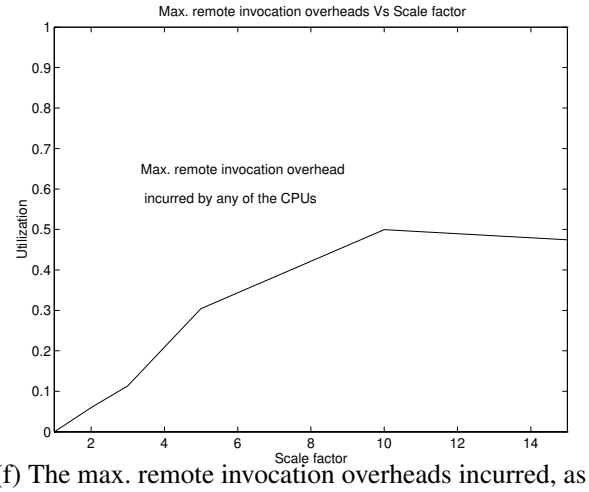
(c) (Optimized) throughput VS the scale factor



(d) The number of (optimized) database and location server replica VS the scale factor.



(e) The maximum CPU utilization and the load imbalance VS the scale factor.



(f) The max. remote invocation overheads incurred, as a function of the scale factor.

Figure 8: Results of the scalability analysis of the call processing system

TABLE 2. Scalability analysis results for the call processing system.

Scale Factor	Optimal no. of replicas		Productivity (Optimized) (ms^{-1} per unit cost)	Scalability metric value (Optimized)	Throughput (Calls per hour)$\times 10^6$	Normalized* Response Time	System Cost (units)
	Database	Location server					
1	1	1	0.1600	1	1.0958	0.72	1.1
2	1	1	0.1492	0.9325	2.1273	0.88	2.1
3	1	6	0.1444	0.9025	3.0531	0.89	3.1
5	4	2	0.0866	0.5415	3.6485	1.16	5.4
10	2	4	0.0589	0.3683	4.4934	1.11	10.2
15	2	10	0.0387	0.2421	4.4934	1.12	15.2
*The response time is normalized to the target mean response time of 10 ms							

The factors which have to be balanced in the optimization of the scalability enablers include:

- Collocation and distribution of software tasks: collocation reduces the remote invocation overhead but may cause a load imbalance in the overall system,
- Replication of the database, which affects load balancing, remote invocation costs, overheads for coordination (and thus also latency) and system cost. (Replication of the location server however does not have a cost or overhead associated with it.)

The *overhead latencies due to database replication*, measured per response, have been summarized below. These results correspond to the optimum replication levels mentioned in Table 2.

	Scale Factor = 5	Scale Factor = 10	Scale Factor = 15
Overhead latency (per response) due to database replication (ms)	2.10	0.7	0.7

The *remote invocation overheads* incurred are different for each CPU; the maximum among them are plotted in Figure 8(f) below.

Overall, the results show that scalability is reasonable up to a factor of 3. Beyond this, the scalability metric degrades, although capacity continues to increase up to a factor of about 10. Beyond this the system is bottlenecked and capacity is saturated.

Some details shown by the Table and by Figure 8 are:

- In Figure 8(a) the available productivity of the system drops gradually up to $k = 3$ and then more steeply. Figure 8(b) shows the scalability metric. It drops to 0.8 at about $k = 4$, indicating scalability up to a factor of 4.
- Figure 8(c) shows the throughput, with a knee at about $k = 3$. At this point it has been increased from about 1.09 million calls/hr to about 3.3 million calls/hr, while maintaining a good QoS. For scale factors beyond 3, the optimization gives a response time a little higher than the target value (which is encouraged by the metric, because it also gives a higher throughput).
- Figure 8(d) shows the replication of the location server and the database server, which follows a generally increasing trend with a lot of variation. The cost-benefit balance of replicas in the middle range is pretty well neutral, so the optimizer has stopped in different parts of the space

in different runs. In fact, at $k = 10$ and 15, the optimizer chooses to leave CPUs unused (one at 10, two at 15) in order to be able to collocate some of the objects and save on remote-invocation overheads. Due to the existence of unused CPUs (one unused CPU for scale factor = 10, and two for scale factor = 15), the maximum CPU utilization is the same as $U_{\max} - U_{\min}$ as seen in Figure 8(e).

- Figure 8(e) gives a picture of the load balance, which becomes worse as the number of CPUs increases, because of constraints on allocation, differences in task demands, and remote invocation costs. At $k = 10$ and 15 the minimum utilization is zero because of unused processors.
- As the software tasks are spread out across more CPUs, they incur increasing remote invocation overheads, as seen in Figure 8(f). The worst overhead percentage among the processors is plotted; it levels off at about 50%.

Summary

Thus, we conclude that the call-processing system is scalable up to a scale factor of 4, at which point it can support about 3.3 million calls per hour. If it needs to be scaled beyond this point in a cost-effective manner, the following improvements could be tried:

- The organization of software objects into concurrent tasks could be redesigned, to make their execution and communication demands more equal.
- The database schema could be modified so the database could be partitioned in various domains rather than replicating it, and the consistency management overheads can be reduced. However, the location service faces an increased execution demand in such a scenario.

The optimization by simulated annealing took about 10 hours (on a SPARC Ultra-1 workstation) for each scale factor. This is entirely practical for a major evaluation, but it is also quite heavy, and a faster optimization technique would be preferred.

8 Conclusions

The proposed *strategy-based scalability* metric generalizes the well-known metrics for scalability of parallel computations, to describe heterogeneous distributed systems. In these systems a uniform increase in all types of components is usually not a reasonable scaling strategy.

The principal new features of this metric are: separating the impact of throughput and response time on the metric, formalizng the notion of a scaling strategy, introducing a quality-of-service evaluation, and introducing formal scalability enablers which are optimized at each scale factor. The metric is the ratio of the system's productivity in a scaled version, to the productivity of a base case. Relating scalability to productivity is consistent with quite general quality-of-service evaluation, and with previous work on metrics for parallel systems.

The previous scalability metrics are special cases. For example, in scalability based on fixed size speedup, the scaling strategy is to use k processors, throughput is the inverse of completion time, cost is k, and the QoS function is $F = 1$. For scalability based on fixed-time speedup, the scaling strategy is also to use k processors but to also change the workload W to a value which keeps the completion time constant. Throughput is now constant, cost is k, and the QoS function is $F = W$.

The new strategy based scalability metric gives reasonable results for a large collection of idealized and well-understood system models, in the form of queueing models suitable for distributed systems. While it requires substantial effort to apply it to real systems the effort is manageable.

The contributions of this work are the new framework (including an open-ended range of possibilities for different quality-of-service evaluation functions, and for different scaling strategies), the new metric, a bounding calculation, and practical numerical techniques for evaluating the metric on real systems. These techniques are applied to two substantial problems which have not been described before, to indicate both the scaling limits and how the scalability might be improved.

The new framework is only applied here using models, to evaluate systems that have not yet been deployed, but it could also be used with measurements to evaluate live deployed scaled systems.

The paper has defined scaling only with a single scale factor, but the framework applies equally to multiple scale factors, describing independent scaling of different attributes of the system. The productivity and scalability would then be defined as functions of a vector k.

Acknowledgments

Discussions with Pankaj Garg and Jerry Rolia were helpful in the early stages of this work.

References

[1] P.P. Jogalekar and C.M. Woodside, "Evaluating the Scalability of Distributed Systems", Proc. 31st Hawaii Int. Conf. on System Sciences, vol. 7, pp. 524–524, January, 1998.

[2] X.H. Sun and L.M. Ni, "Scalable Problems and Memory-Bounded Speedup", J. of Parallel and Distributed Computing, vol. 19, pp. 27–37, 1993.

[3] X.H. Sun and J. Zhu, "Performance Considerations of Shared Virtual Memory Machines", IEEE Trans. on Parallel and Distributed Systems, vol. 6, no. 11, pp. 1185–1194, November 1995.

[4] A.Y. Grama, A. Gupta and V. Kumar, "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures", IEEE Parallel and Distributed Technology, pp. 12–21, August 1993.

[5] S.R. Sarukkai, P. Mehta and R.J. Block, "Automated Scalability Analysis of Message-Passing Parallel Programs", IEEE Parallel and Distributed Technology, winter 1995, pp. 21–32.

[6] A. Sivasubramaniam, U. Ramachandran and H. Venkateswaran, "A Comparative Evaluation of Techniques for Studying Parallel System Performance", Technical Report GIT-CC-94/38, College of Computing, Georgia Institute of Technology, Atlanta, September 1994.

[7] O. Char, C. Evans and R. Bisbee, "Operating System Scalability: Windows NT vs. UNIX", Intergraph Corporation, available at `http://www.ingr.com/ics/wkstas/ntscale.html`

[8] C. Allison, P. Harrington, F. Huang and M. Livesey, "Scalable Services for Resource Management in Distributed and Networked Environments", WARP Report W1-96, Division of Computer Science, University of St. Andrews, UK. Available at `http://www.warp.dcs.stand.ac.uk/warp`

[9] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson and D. Culler, "Using Smart Clients to Build Scalable Services", Internal report, Computer Science Division, University of California, Berkeley. Available at `http://www.now.cs.berkeley/edu/SmartClients`

[10] Fahim Sheikh, Jerome Rolia, Pankaj Garg, Svend Frolund, Allan Shepherd, "Performance Evaluation of a Large Scale Distributed Application Design", World Congress on Systems Simulation, Singapore, September 1997.

[11] P.P. Jogalekar and C.M. Woodside, "A Scalability Metric for Distributed Computing Applications in Telecommunications", Proc. 15th International Teletraffic Congress - Teletraffic Contributions to the Information Age, eds. V. Ramaswami and P. Wirth, vol. 2(a), pp. 101–110, 1997.

[12] A. Giessler, J. Hanle, A. Konig, E. Pade, "Free Buffer Allocation - An Investigation by Simulation", Computer Networks, pp. 191–208, 1978.

[13] E.D. Lazowska, H. Zahorjan, G.S. Graham, K.C. Sevcik, "Quantitative System Performance - Computer System Analysis Using Queueing Network Models", chapter 5, Prentice-Hall, Inc., New Jersey, 1984.

[14] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", IEEE Trans. on Computers, vol. 44, no. 1, pp. 20–34, January 1995.

[15] J.A. Rolia and K.C. Sevcik, "The Method of Layers", IEEE Trans. on Software Engineering, vol. 21, no. 8, pp. 689–700, August 1995.

[16] P.J.M. van Laarhoven and E.J.L. Aarts, "Simulated Annealing: Theory and Applications", D. Reidel Publishing Company, Boston, 1987.

[17] L. Ingber, "Simulated Annealing: Practice Versus Theory", J. Mathl. Comput. Modelling, vol. 18, no. 11, December 1993.

[18] G.L. Bilbro and W.E. Snyder, "Optimization of Functions with Many Minima", IEEE Transactions on Systems, Man and Cybernetics, vol. 21, no. 4, July/August 1991.

[19] S. Majumdar, C.M. Woodside, J.E. Neilson and D.C. Petriu, "Performance Bounds for Concurrent Software with Rendezvous", Performance Evaluation, vol. 13, pp. 207–236, 1991.

[20] "Generic Functional Architectures for Transport Networks", International Telecommunications Union Recommendation no. G.805, November 1995.

[21] P. Trantafiliou and D.J. Taylor, "The Location-Based Paradigm for Replication: Achieving Efficiency and Availability in Distributed Systems", IEEE Trans. on Software Engineering, vol. 21, pp. 1–18, January 1995.

[22] A.M. Pan, "Solving Stochastic Rendezvous Networks of Large Client-Server Systems with Symmetric Replication", Master's thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, September 1996.

[23] T. von Eicken, A. Basu, V. Buch and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", Proc. 15th ACM Symposium on Operating Systems Principles, Colorado, pp. 1–14, December 1995.

[24] J.C. McDonald, "Fundamentals of Digital Switching", Chapter 4, Plenum Press, New York, 1990.