Self-Organizing Network Services

Hani Jamjoom, Sugih Jamin, Kang Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109
{jamjoom, jamin, kgshin}@eecs.umich.edu

Abstract— **There is a proliferation of application-level servers that offer enhanced network services beyond the simple packet forwarding provided by the underlying network infrastructure. Examples of such servers range from Web server mirrors, Web caches, Web page distilling proxies, video transcoders, and application-level multicast routers, to application-level load-adaptive multipath routers. A fundamental question arising from the deployment of such servers is where to place them within a network. This paper explores technical issues related to the creation of an infrastructure to allow self-organization of network service placement based on observed demand for each service. In so doing, we propose a framework, called Sortie, whereby services are allocated on network nodes based on a set of very simple rules independently executed by each node. The distributed nature of allocation decisions ensures the scalability of the framework. We also present simulation results confirming the stability and efficiency of the proposed framework.**

Keywords— **Network services, self-organization, demand-triggered**

I. Introduction

Along with the growth of the Web and the increasingly prevalent use of the Internet as an infrastructure to support multimedia applications, there is an increasing demand for distributing application-level services across the Internet. This ranges from mirroring of Web content and multi-player game servers to deployment of distilling proxies and video transcoders. Assuming there is a network of nodes that are willing to host such services, a natural question arises: where and how many replicas of these services should be made available? A non-trivial answer to this question would ensure that a minimal amount of network resources is consumed by the resulting service allocation.

Self-organization has been proposed as a scalable means of distributing and managing these services. We define *self-organization* as the ability of a service to replicate and remove instances of itself based on dynamically-fluctuating demand. The use of self-organization has been explored in managing video transcoders [1–3], web content mirrors [4], and web caches [5, 6]. Each of these efforts explores the use of self-organization in its *specific* application context. In contrast, we propose a *general* service whereby application services are automatically replicated and strategically placed at participating nodes in a network based on usage patterns. Such a service should (i) provide relatively fast adaptivity in replica demography when usage pattern shifts, (ii) prevent oscillation when usage pattern is not at equilibrium, and (iii) maintain stability when usage pattern is at equilibrium.

Specifically, we propose and evaluate a framework for self-organizing network services which we call *Sortie (Self-ORganizing acTIve sErvices)*. Sortie running at each participating network node measures the local demand for each network service independently of all other nodes. When demand for a service reaches a high watermark, the service becomes a candidate for local replication; should demand drops below a low watermark, the service may be removed. We will in Section III-D describe an exponential back-off scheme to dampen oscillation in replica population when usage pattern is not at equilibrium or when usage pattern oscillates around the equilibrium. While our work is intended for intranets, the distributed nature of its allocation decision makes it extendable to the Internet.

Aside from replica placement, such a framework must also provide a protocol for the advertisement and discovery of service replicas. We do not propose such protocol. Instead, we adopt recent work on an intentional naming architecture [7] or on resource discovery [8–10] for providing the necessary functionality.

We elaborate on the goals of Sortie in the next section. Section III zooms on the underlying design principles and architecture of Sortie. In Section IV we present results from simulations we conducted to assess the effectiveness of the architecture. Aside from providing initial confirmation on the stability of service replica population managed by Sortie, the results also indicate that the service allocation is efficient in its network resource consumption. Since finding the optimal solution to the placement problem reduces to the NP-complete bin-packing problem, Sortie cannot hope to achieve optimal placement.[1] As the results show, however, Sortie does provide a tuning parameter to trade off the efficiency of various resource consumptions. After presenting related work in Section V, we close this paper with a conclusion and a set of open issues in Section VI.

II. Sortie Goals

Different applications may have different needs from a self-organizing framework such as Sortie. Instead of trying to provide an exhaustive list of all encompassing goals, we discuss below only a handful of goals we consider fundamental to all self-organizing frameworks.

Stable Adaptability. The first issue to consider is whether the demography of services managed by Sortie is stable. When node resources are abundant, replicating a service may be simple; if resources are scarce, on the other hand, activating a service at a node may require downloading a copy of the service from a remote site, which will itself consume network resources. In either case, there is an overhead to activating a service, at the very least there is the time lag between service activation and its use: the service must first be advertised and packets routed to the node where the service resides. A desirable behavior is that replica demography will reach an equilibrium where few further replications are required. Unstable replica demography is comparable to the virtual-memory thrashing problem found on single-node systems.

Since Sortie's decision to keep or remove a replica is based on measured demand for the service, two factors determine the stability of the system. The first is the length of demand history. The longer the history is kept, the more stable the system may be. However, if the demand pattern changes rapidly, the longer the history is kept, the slower Sortie will converge to the new set of services required by the new demand pattern. The second factor determining the stability of the system is the demand pattern itself. If the demand pattern is very bursty, or if demand oscillates around equilibrium, there could be some oscillation in replica demography. A self-organizing framework must thus have a second-level control mechanism to adjust its adaptivity and dampen oscillation. Ideally, this second-level control mechanism could automatically adjust to observed behavior of replica demography; at the very least it must provide tuning knobs network administrators can set manually.

To ensure stability when resources are scarce, or in the presence of oscillating demand patterns, Sortie may opt to maintain the replicas of only a subset of services. We assume that each service will have a "home" node where the service will always be available. Hence if the service cannot be replicated across the network, packets requesting the service will have to be routed towards the services' home node. This will maintain stability at the cost of decreased route efficiency.

Route Efficiency. We call the path a packet takes when it doesn't require any application-level service the *packet's original path*.[2] In the ideal case, a packet that requires a service will find the service along its original path. If a large percentage of packets must deviate from their original paths in search of the required services, the service placement is said to have poor *route efficiency*. When resources are abundant, route efficiency may be kept high by liberal replication of services. When resources are scarce, Sortie may have to trade off route efficiency for stability. Viewed another way, inefficient routing may be indicative of the need to add more resources on the network. Since Sortie measures service demand at each participating network node, it can help pinpoint resource bottlenecks.

Placement Efficiency. In evaluating the efficiency of a given service placement, we make a distinction between route efficiency and placement efficiency. We defined the route efficiency in terms of deviation from the original path. In contrast, to evaluate placement efficiency, we take application-level semantics into account. That is, factoring aside route efficiency, *placement efficiency* considers the question: where along a chosen route should a requested service be applied? We take factors such as bandwidth requirements before and after service application and node resource availability into account when answering this question.

For the purpose of illustration, let us consider web page distilling and multicast delivery as two example application-level services. Web page distilling reduces the amount of bandwidth required for transmission of a Web page by reducing the amount of information contained in the page. It achieves the maximum amount of bandwidth reduction if applied at the source, either at the original web server or one of its mirrors. If the source does not support distilling, then the closer to the source distilling can be performed, the larger the potential bandwidth saving. On the other hand, multicast delivery increases bandwidth usage at multicast dispersion points,[3] hence to reduce bandwidth demand, this service should be applied as close to the destinations as possible.

[1] Optimal placement for closed small-size networks may be computable in bounded time.

[2] The original path may not be the shortest path, due to policy routing.

[3] Only a single copy of the packet needs be sent on broadcast networks.

To achieve scalability, a general self-organizing framework cannot hope to evaluate the data-handling semantics of every service. Take web caching for example. To reduce download latency and bandwidth usage, one may want to place caches as close to clients as possible. This, unfortunately, lowers the aggregation of requests each cache sees. Hence caches placed some distance away from clients may actually result in a higher hit rate, providing lower latency and bandwidth usage overall. Instead of building in intelligence to learn data-handling semantics of every service, our approach in Sortie is to associate a *rate transformation factor* (α) with each service. How a service provider can use this service attribute to control Sortie's behavior is examined in the next section.

With these goals in mind, we now discuss the design principles and architecture of Sortie in greater details.

III. Framework Design

The authors of [11–13] have observed that complex, and often unintended, dynamics can arise from seemingly innocent behavior of algorithms deployed on the Internet. To avoid such unintended dynamics, we have followed a design principle in Sortie whereby each node running Sortie executes only a small set of well-defined rules. Researchers studying natural and biological systems, cellular automata, and artificial life have also noted how sustainable large complex systems often arise from simple, regular components, each executing a small set of simple rules. By following a similar design precept, we show that the set of simple rules we have designed for Sortie will lead to a sustainable self-organizing network.

A. Self-Organizing Algorithms

This section outlines five different approaches to building a network of self-organizing service replicas. The approaches considered are required to satisfy some well-known principles of scalable design such as decentralized decision-making based on local information, use of soft states, being robust to unreliable information on network states, being adaptive to changing network conditions, etc. Hence in all of the approaches studied, each network node measures only *local* demand for each service and makes *independent* replication decisions based on its own network observations. As we mentioned in the Introduction, the replication algorithm aside, a framework for self-organizing network services must also incorporate a protocol whereby service replicas can be advertised to all participating nodes. Each node must also have an underlying packet-forwarding substrate that can differentiate between packets not requiring services other than fast packet forwarding from those that require additional network services.[4] In our description of the five self-organizing replication algorithms below, we assume the availability of such service-advertisement protocol and packet-forwarding substrate in all cases.

We define a taxonomy of self-organizing algorithms along two dimensions: (1) when to replicate a service, and (2) where to apply a service. A service can be replicated as soon as there is a demand for it, or it can be replicated only if the demand level reaches a high watermark. We call the former *Greedy* replication and the latter *Miser* replication. Orthogonal to the demand level, a service can be replicated at a node close to the source, or it can be replicated close to the destination. We call the former *Eager* replication, and the latter, *Lazy* replication. Hence there are four possible algorithms: (1) Greedy/Eager, (2) Greedy/Lazy, (3) Miser/Eager, and (4) Miser/Lazy.

Finally, Sortie uses a Miser algorithm that takes the application-level rate transformation factor (α) into account when deciding where to replicate a service. The replication rules that Sortie uses are: (1) a service is replicated locally only if demand for it reaches a high watermark, and (2) replication decisions consider the service's rate transformation factor (α). We describe in the following section the second rule in more detail.

When a service is first introduced on the network, it is assumed to reside only on its *home* node(s). We require each service to have one or more home nodes from which it cannot be automatically removed. The home node(s) must be participants in the self-organizing framework. The existence of a new service will be advertised to other nodes running Sortie. Each node then monitors local demand for the new service and executes Sortie's replication algorithm. Once a node acquires a replica of the service, it proceeds to advertise the availability of the service. The node continues advertising the availability of the service periodically until the replica is removed (presumably to make room for a replica of another service).

B. Service Application

Two mechanisms work in conjunction with each other to achieve self-organization. The first is deciding whether to apply the requested service locally or forward packets to another switch for service. The second is deciding when to replicate a service. These decisions, while made independently by each switch, impact the demand for services that is seen by other switches and can thus affect replica placement in the network. In this subsection we describe

[4] In the former case, packets should be forwarded along the forwarding code's fast path and not suffer additional performance degradation.

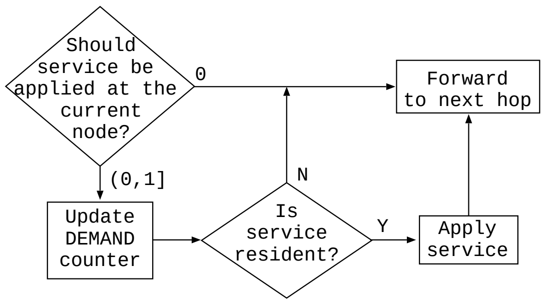


Fig. 1. Service application flowchart. The returned value from the top-left decision box is used to affect the demand for the service.

the service application process; we defer description of the replication process to the next section.

In general, once a packet is determined to require an additional network service other than fast packet-forwarding, it is passed from the underlying forwarding substrate to Sortie. Sortie first determines if the required service is best applied at the current node or not. If the service is not best applied at the current node, it looks up the service's "routing table," determines the "next hop" towards the destination, and forwards the packet to the next hop.

```
Miser/Eager:
    return $D_k^d/(D_s^k + D_k^d)$

Miser/Lazy:
    $r = D_s^k/(D_s^k + D_k^d)$
    if (¬ (∃ downstream node that can
        participate in self-organization))
        return 1.0
    else if (¬ (∃ service on shortest path))
        return $r$
    else
        return 0.0

Sortie:
    if $\alpha < 1$
        use Miser/Eager
    else if $\alpha > 1$
        use Miser/Lazy

where:
  $D_i^j$: distance between node $i$ and node $j$,
  $s$: source node,
  $d$: destination node,
  $k$: current node,
  $\alpha$: rate transform factor,
  $\kappa$: tunable parameter, $\kappa \in$ (0,1).
```

Fig. 2. Algorithm to determine if a service should be locally replicated for Miser replications.

The five self-organizing algorithms differ in deciding when to apply the service. We only focus on Miser replications since the Greedy ones have a trivial decision process. The three Miser replication algorithms share a common decision process summarized in Figure 1. The fractional value is used to affect the demand for the service. A higher value implies that the service is more suited in the current switch, and vice versa (more on this will be discussed in the next section).

The three Miser algorithms differ in deciding where to apply the service. Figure 2 outlines this decision process for the Miser replications. Miser/Eager depends on the ratio of the distance (D_k^d) between the current node (k) and the destination (d) of the packet and the path length (D_s^d) from source (s) to destination (d). The definition of "distance" is specific to each application-level service. In the simplest case, the distance is the number of hops stored in the routing table of the network. Services requiring accurate distance may have to consult network topology services [14, 15]. Multicast application may define "distance" as a function of the number of outgoing interfaces a packet must be delivered through.

In contrast to Miser/Eager, Miser/Lazy depends on the ratio of D_s^k to D_s^d. Furthermore, Miser/Lazy depends on the availability of service in downstream nodes. Only when the service is not available on the original path, Miser/Lazy returns a non-zero value. This tries to concentrate the demand at the nodes where packets deviate from the original path and eventually causes the replication of services at these nodes.

Sortie uses an *adaptive* combination of the two algorithms. If the service is best suited near the source (destination) it uses Miser/Eager (Miser/Lazy). It relies on services' rate transformation factor (α) to decide which algorithm to use. This rate transformation factor is a service-specific attribute. It specifies the expected rate change after a data stream has been transformed by the application of the service. A service with $\alpha < 1$ *reduces* the data rate after transformation. Web distilling services, for example, have $\alpha < 1$ and should be placed as close to the source as possible. In contrast, a service with $\alpha > 1$ should be applied near the destination(s). For some other services, α may be unity, i.e., the data rate after transformation is the same as before the application of the service. Placement of such services does not affect network bandwidth, and thus these services could be placed *anywhere* along the path where demand for that service is high. By using α to switch between Miser/Eager and Miser/Lazy, Sortie combines the benefits of both algorithms while improving the placement of services.

C. Service Replication and Removal

The decision of the algorithm in Figure 2 is given as a fractional value (r) between 0 and 1. A value of 0 means the service is not to be applied at the current node. When the value is non-zero, Sortie keeps track of user demand for services that would benefit from being located at the current node. User demand is estimated by keeping an exponential moving average of the service request inter-arrival times:

$$a^i_{new} = wa^i_{old} + (1 - w)r\delta^i_t, \qquad (1)$$

where w is the exponential averaging weight, v is the fractional value returned by the decision process described in the previous section, δ^i_t is the inter-arrival time of requests for service i, a^i_{new} is the new average inter-arrival time of requests for service i, and a^i_{old} is the old average inter-arrival time of requests for service i.

Based on the demand for each service, Sortie decides whether to replicate the service or not. When the demand exceeds a threshold watermark, the service becomes a candidate for replication. To better utilize switch resources, Sortie tries to replicate services that would maximize the total number of served packets and at the same time minimize the total number of packets deviating from the shortest path. To do so, the watermark is adjusted dynamically according to the demand of other services. It is set to the lowest demand of services that are currently allocated. The Sortie's replacement policy can be thought of as *least-frequently-used* replacement.

In describing a node's management of replicas, we use the terms "remove" and "delete" to mean "stop advertising" and "physically delete from node," respectively. When a node removes a replica, it does not immediately delete the replica. If there is no contention for a node's resources, it retains the replica but discontinues the advertisement of its existence. Hence, if it ever becomes necessary for the node to host the service again, a replica will already be resident in the node. A replica is deleted only when there is contention for the resources it holds.

D. Exponential Back-off

One of Sortie's main goals is to minimize oscillation in replica population. Oscillation can occur even under constant traffic patterns. This is especially true for Greedy algorithms when switch resources are lower than those required by the demanded services. Miser algorithms offer better stability against oscillation since they give a high-demand stream priority over lower-demand ones. However, the switch may falsely interpert traffic bursts as an indication for high service demand causing it to replicate the service.

```
if (service removed longer than $I_{th}$ duration ago and
   there has been no requests during $I_{th}$)
      Replicate service
      $R_i$ = 1
      $Z_{th}$ = $I_{th}$
else if (service frozen)
      if (frozen for more than $Z_i$ period)
            Replicate service
      $R_i$++
else service is not frozen and was recently removed
      Freeze service
      $Z_i$ = $2^{R_i} * Z_{min}$
      if ($Z_i > Z_{th}$)
            $Z_{th}$ *= 1.5
            $R_i$ = 0

where:
   $I_{th}$: idle time threshold
    $R_i$: replication count of service $i$
    $Z_i$: service $i$'s freeze duration
 $Z_{min}$: minimum freeze duration
  $Z_{th}$: freeze duration threshold
```

Fig. 3. Exponential back-off algorithm to reduce oscillation in replica population.

When the demand pattern changes rapidly, one way to dampen oscillation in service replication is to keep a longer demand history, or equivalently, raise the high watermark. Raising the high watermark, however, reduces the adaptivity of Sortie since it only considers high demand patterns. To improve Sortie's responsiveness, we could set the high watermark based on observed frequency of demand pattern changes. Or we could use the observed frequency of replication requests directly in the replication decisions. We chose to do the latter in Sortie by freezing service replication when the demand pattern is too bursty. This is an *exponential back-off (EB)* algorithm since the freezing period is exponentially increased for each replication request during an oscillation period.

Figure 3 formally defines the exponential back-off algorithm of the freeze duration. Each service i has associated with it a freeze duration (Z_i). A service is frozen if a request for its replication arrives earlier than a preconfigured idle time threshold (I_{th}) since its last removal. I_{th} is used to approximate the idle time between two streams. A frozen service cannot be replicated until its freeze duration expires. Every time a service is frozen, its freeze duration is exponentially increased. To ensure Sortie's responsiveness once the demand pattern is no longer oscillating, we reset Z_i to the minimum value if it becomes larger than a threshold (Z_{th}). The threshold (Z_{th}) itself is

Network Characteristics	
# of nodes	91
AGGvg. hop count	5.2
Network diameter	9
# of switches	15
Avg. degree of switch connectivity	3
# of edge nodes	10
Avg. # of nodes per edge switch	6.6
# of services	10

TABLE I
Characteristics of the simulated network.

Characteristics	High Traffic	Low Traffic
Peak rate (Kbps)	28.8	14.4
Avg. on time	10	10
Peak/avg. ratio	2.0	2.0
On shape	1.1	1.1
Off shape	1.1	1.1
Packet size (bytes)	1500	1500

TABLE II
Characteristics of Pareto sources.

increased by half again every time Z_i reaches it. We will analyze the effects of Z_{min} and I_{th} in Section IV-B.5.

IV. Experimental Evaluation

We focus our evaluation on the placement and stability of service replicas. This section is divided into two parts. We first describe the simulation environment, evaluation metrics, and underlying assumption; we then discuss our findings in greater detail.

A. *The Simulation Environment*

We focus our attention on a single autonomous system (AS) since it contains a tractable number of switches, and hence it is easier to demonstrate the key features of Sortie. (Interactions between multiple ASs will be addressed in our future work.) In an AS, each switch will be informed of the replication and removal of every service in the system. This way, each switch can keep track of existing services and can quickly forward a packet to the closest switch when the service required by the packet is not available locally.

To focus on our primary objective, we simplify the allocation problem by only considering resources of equal constraints. This transforms the problem into a tractable operation of swapping one service with another. Otherwise, it is NP-Complete since it easily maps to the well-known bin-packing problem.

The performance of the five placement algorithms in Section III-A is evaluated using simulations. Two main assumptions are made to simplify the implementation process. First, that service advertisement is implicit. That is, there is zero propagation delay between when a service is advertised or removed at a switch and the time the information reaches other switches. We do this to avoid instabilities caused by the lag time between removing a service and informing other routers of this action. We also do not model the process of replicating a service. Instead, we keep track of the number of replication requests. As a way of relaxing the second assumption, in Section IV-B.4 we separately evaluate the effects of replication delay on the performance of the five algorithms.

The network used for our simulation study is randomly generated. Table I summarizes the characteristics of the network. The core switches are randomly connected using Waxman's [16] model. In our simulations, a source sends requests for certain services using a Pareto distribution. The (source, sink) pair of each connection are chosen at random, using uniform distribution. We model two traffic levels, high and low, as described in Table II. We choose to use Pareto distribution with a high variance (i.e., the shape values are close to 1) to better approximate the behavior of host traffic. Because of the large number of performed simulation trails and to allow our simulation to complete in reasonable time, we only choose to mimic hosts with low rates, similar to those connected via modems.

For each of the five placement algorithms studied, we ran each experiment 50 times for 600 seconds of simulation time. Each run is parameterized by (1) service requested by sources, (2) the amount of resources at each switch, (3) the rate-transformation factor of each service, (4) the demand level of each connection, and (5) the delay of replicating a service. The following subsections present and analyze the results of our simulation.

B. *Simulation Results*

Three metrics are used to compare the five placement algorithms: stable adaptability, route efficiency, and placement efficiency. We then discuss the effects of replication delay, and finally the effectiveness of exponential back-off.

B.1 Stable Adaptability

We measure system stability in terms of the rate of replication requests because a stable system will reach a point where no further replication requests are issued for a stable demand pattern. As discussed earlier, the burstiness of the sources may result in continual replication requests causing replica oscillation. So, Algorithm A is said to be more stable than B if it generates fewer replication requests.

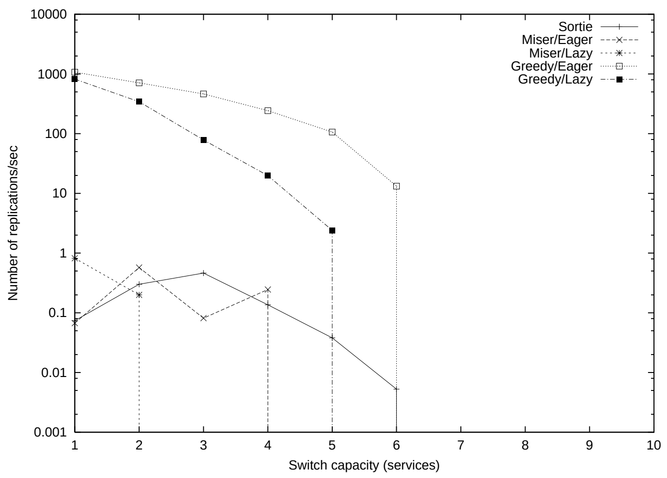


Fig. 4. Rate of replica requests. This figure plots the *average* rate of replication as a function of switch capacity. The vertical drops are due to the logarithmic scale of the y-axis. It should be read as the function having zero value after the drop.

Figure 4 shows the rates of replication requests that are generated by all switches for the five placement algorithms. In each case, an edge switch gets requests for an average of 7 different services from the sources that are connected to it. When switch resources are varied from accommodating 10 services to only 1 service, Miser replications, which includes Sorite, showed substantial improvements in system stability over the other non-self-organizing, Greedy, techniques. Once switch capacity can accommodate 7 services, all of the algorithms have enough resources to host all of the requested services and are thus stable.

Figure 4 also shows that Sortie is less stable than the other Miser algorithms. This is because Sortie attempts to place services according to their rate-transformation factor, thus focusing on a smaller subset of switches and making it more susceptible to oscillation.

B.2 Route efficiency

We measure route inefficiency by counting the number of packets that deviated from their original paths at each switch over the total number of packets forwarded by the switch. A large percentage of packets not following their original paths will result in poor utilization of network resources. Figure 5(a) shows the long-term average route inefficiency as a function of switch capacity. As the amount of resources decreases, route efficiency deteriorates. This routing inefficiency experienced by the Miser algorithms is offset by the improved stability of service replica.

To complement our route efficiency measurements, we compute the degradation in the average hop count that is experienced by each of the five replication algorithms. This computation reflects the amount of deviation that packets make. That is, as the amount of deviation increases, so does the degradation in the average hop count. Our simulation showed that Miser replications suffer less than 5% degradation when switches can accommodate more than 2 services. Greedy replications have perfect route efficiency because packets do not deviate from their original paths. However, as we have seen in Section IV-B.1, they are less stable than the Miser algorithms.

Since our simulation does not model the concept of home switches, when the requested services require more switch resources than what is available, packets requesting a service may not get served by any of the switches they traverse. We call this *service starvation*. In Miser replication, services with low demand are more prone to starvation since high-demand ones are given higher priority in replicating services. From the standpoint of individual packets, service starvation is undesirable since in reality packets must be forwarded to the home switch, thus reducing route efficiency. However, denying requests for services with low demand maximizes the total utilization of the network since a greater number of packets will be serviced. Figure 5(b) shows that Miser replications starve less than 1.5% of the traffic when switches have a moderate amount of resources. Section IV-B.4 shows that stream starvation becomes a more critical issue when the replication delay is considered.

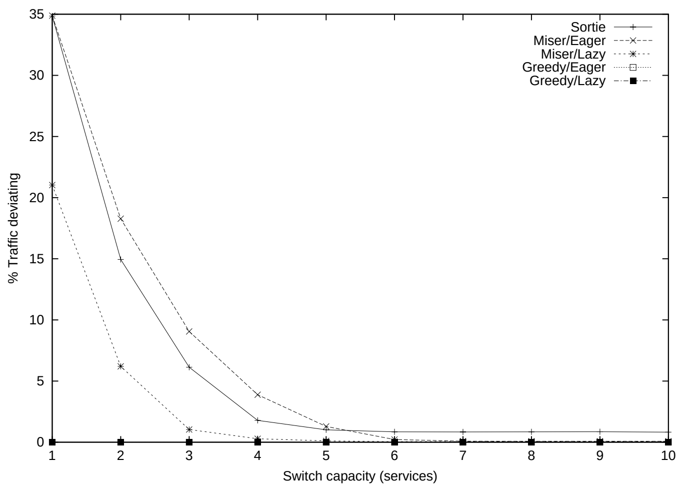
B.3 Placement Efficiency

From the above analysis, ignoring the rate-transformation factor (Miser/Lazy replication) seems to perform best all the time. This is not the case, especially when we analyze the efficiency of service allocation for the five algorithms. We measured replica placement efficiency with an average number of hops a service is placed away from the traffic source (destination) if the service's rate-transformation factor is < 1 (> 1). Since we are averaging over all services, we can compute placement efficiency using the following equation:

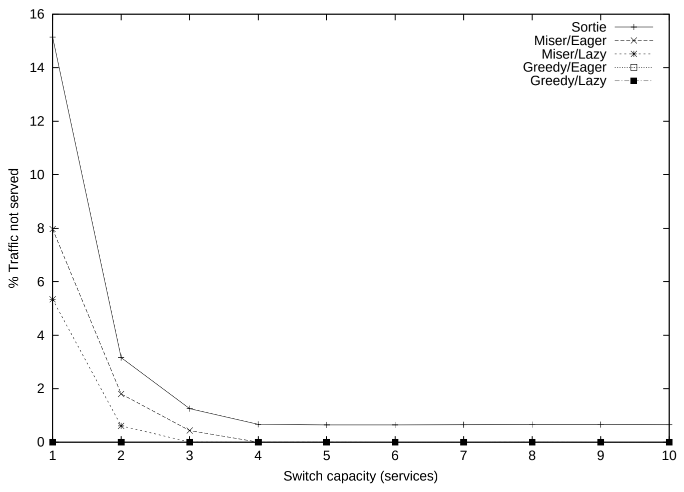
$$\text{Placement Efficiency} = \frac{1}{n}\sum_{k=1}^{n} h(k) \qquad (2)$$

where n is the number of packets, and $h(k)$ is the number of hops after packet k got served if $\alpha > 1$, or before packet k gets served if $\alpha < 1$.

We plot placement efficiency along two dimensions: (1) as a function of switch capacity and (2) as a function of service distribution according to α. Figure 6(a) plots the average placement efficiency of the five algorithms as a

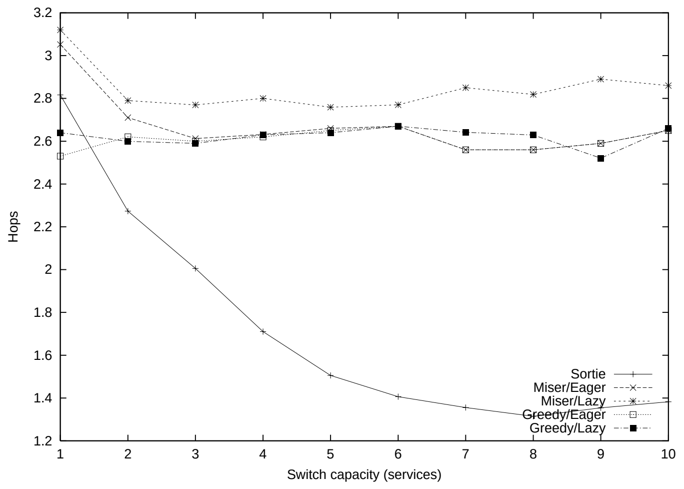


(a)

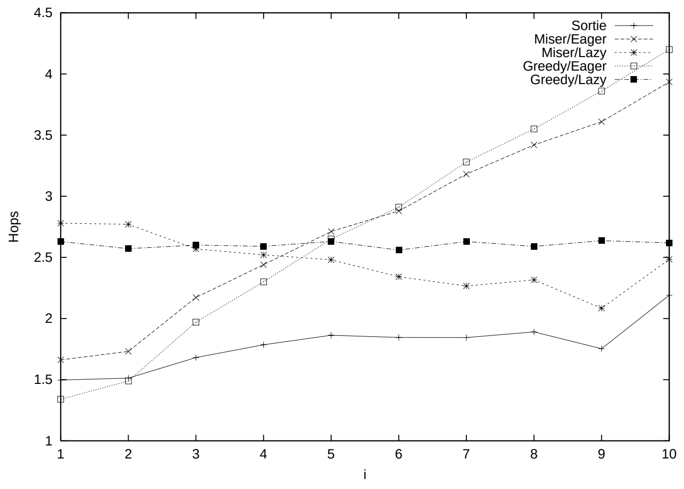


(b)

Fig. 5. Route efficiency: (a) plots the percent of traffic deviating from the original path; (b) plots the percentage of traffic not served as a function of switch resources.



(a)



(b)

Fig. 6. Placement efficiency. (a) shows the relative distance of services as a function of switch capacity. Services are distributed such that 50% of services have $\alpha < 1$ and 50% have $\alpha > 1$. (b) shows the relative distance of services according to α. i (on the x-axis) is the number of services (out of 10) that have $\alpha > 1$ and $(10 - i)$ is the number of services that have $\alpha < 1$.

function of switch capacity while assuming 50% of services have $\alpha < 1$ and the remaining 50% have $\alpha > 1$. The figure shows that Sortie's placement decisions are improved with the increase of switch resources. The other algorithms have placement efficiency close to the center of the network. This is expected because of the uniform distribution of the (source, destination) pairs. The small differences between the five placement algorithms are due to the relatively small diameter of the network.

In Figure 6(b) we fix switch capacity to accommodate 3 services and change the distribution of services according to α. The figure shows that the performance of the five algorithms is a function of such distribution. When all services are best applied near the source, Greedy/Eager produces the best result since it always serves packets at the first router from the sources. In contrast, since Miser/Lazy tries to delay application of the service as late as possible, it performs worst. In nearly all cases Sortie outperforms the other four algorithms.

B.4 Effects of Service Replication Delay

In the previous analysis we have assumed that replication of a service incurs negligible cost. In reality replicating a service can incur a significant delay. This section analyzes the effect of replication delay on the performance of the five placement algorithms.

	Delay (sec)			
Algorithm	0.0	0.1	1.0	10.0
Sortie (%)	1.1	0.7	1.4	32.4
Miser/Eager (%)	0.004	0.7	1.5	35.5
Miser/Lazy (%)	0.003	0.6	1.3	27.4
Greedy/Lazy (%)	0.0	21.9	49.3	62.6
Greedy/Eager (%)	0.0	47.6	69.7	82.8

TABLE III
Effect of replication delay on starved packets.

	Delay (sec)			
Algorithm	0.0	0.1	1.0	10.0
Sortie (%)	3.0	5.7	7.2	52.2
Miser/Eager (%)	1.3	9.4	12.3	56.6
Miser/Lazy (%)	0.2	2.8	4.9	42.2
Greedy/Lazy (%)	0.0	0.0	0.0	0.0
Greedy/Eager (%)	0.0	0.0	0.0	0.0

TABLE IV
Effect of replication delay on route efficiency.

We only model the time delay in replicating a service and not any additional network traffic generated by it. We also do not buffer packets at switches primarily because of the wide variation of replication delay. Instead, we assume that the switch will forward packets to the closest switch that has the desired service. Buffering packets is suitable for small replication delays or slow traffic sources. It has the advantage of improved network utilization, i.e., a fewer number of packets deviating from their original paths. On the other hand, it will cause packet loss when buffers overflow. In our simulation, we use the forwarding technique for two reasons. First, it simplifies resource allocation at each switch. Second, we can observe the effects of various time delays on our performance metrics.

Table III shows that replication delay has a direct effect on the percentage of starved packets, especially for the Greedy algorithms. This is mainly due to the high number of requests that both Greedy algorithms issue at the same time without considering an alternative location while the desired service is being replicated. Replication delay also has an effect on route efficiency (Table IV) which is especially apparent in the Miser algorithms since they route packets to the closest alternative switch when the desired service is not available.

What is important is that having large oscillations will cause switches to be under-utilized. Depending on the ratio of time spent replicating a service to the time serving packets, switches can spend a large portion of time replicating services instead of serving packets.

One interesting result is that when the delay is very large, all of the algorithms become unstable. The instability is caused by slow reaction to demand fluctuations. There are two important points to make about this result. First, the need for exponential back-off is more pronounced when the delay is large. The second is that this result emphasizes the importance of stability even at the expense of less than optimal routing efficiency. Because of the poor stability of the Greedy algorithms, once we incorporated replication delay, they produced a large number of starved packets rendering them highly undesirable.

B.5 Exponential Back-off (EB)

In this section we analyze the effectiveness of the EB algorithm as well as the effects of the tuning parameters on its performance. As we will show, exponential back-off dramatically reduces the amount of oscillation in the system. Its effectiveness depends on the level of oscillation as well as on its two tuning parameters Z_{min} and I_{th}. What is important about this technique is that it has minimal impact on Sortie's placement efficiency while still being responsive to demand fluctuations. Because EB maintains a history of the replication of a service, it is both more responsive and just as effective as having a long demand history.

To thoroughly evaluate EB, we focus on a small network composed of a single router with multiple sources sending data to a single sink. We vary the number of sources in the network to change the amount of oscillation experienced by the router. The Greedy replacement policy is used since it is the least stable of all algorithms making it easier to observe the effects of EB.

EB provided approximately two orders-of-magnitude reduction in number of replications in the case of 5, 10, and 20 sources. While it dramatically reduces the amount of oscillation, it also reduces the number of packets that are served by the router. In the case of 10 sources for example, 70% of the traffic is not served, which is expected since only 3 of the 10 requested services will be available at the router. Of course, the EB algorithm can be tuned to be more or less aggressive at reducing the amount of oscillation depending on the cost of replication.

There are two tunable values that EB algorithm uses: I_{th} and Z_{min}. The long-term effects of both values can be observed directly from the actual algorithm in Figure 3. Z_{min} has minimal effect on the long-term performance of the algorithm. However, Z_{min} does have an effect on the responsiveness to changing traffic patterns. While a large Z_{min} will reduce the amount of oscillation very quickly, it will also react slowly to changing demands for services.

The idle threshold value, on the other hand, has a larger effect on the performance of the EB algorithm. This value, which is used to reset the freeze duration after an idle period, is a heuristic for distinguishing between the start and end of different flows (a flow in this context is defined as the traffic generated by an application, e.g., video conferencing session). When it is small, the algorithm assumes any idle period as a sign for a new flow and thus resets the freeze value. On the other hand, if this value is large then the algorithm will slowly respond to changes in user traffic.

Good values for these two parameters depend on the nature of the services and user traffic. The minimum freeze period should approximate the burstiness of the sources while the idle threshold should be chosen to reflect the expected length of a communication session. Z_{min} should also be larger than the expected replication time for a service. This way, once a service is replicated it will be kept for some time before being removed to improve the utilization of router resources.

V. Related Work

We are not aware of any effort on protocols to automatically replicate and distribute new general-purpose in-flight services on a network. Reference [1] provides a general framework for deploying services in the network. Their approach still relies on a client/server approach in which a client instantiates a service request from the Host Managers (HMs) for an application-level service agent, called servent. The request is submitted to the HMs using multicast which in turn causes a HM to create a servent on its local host. The HMs use multicast damping to avoid redundant instantiation of the same servant. The HMs are generally deployed within a cluster environment in which they use a birth-death process to control their population. While their framework is a novel attempt to incrementally incorporate services in local area networks, we focus on creating a global framework for service allocation. We also concentrated on optimal placement within a network for state-less services. This way, reordering service can be achieved to maximize network utilization as well as packet delivery delay.

Recent work by [7] presented a design of an intentional naming architecture in which applications describe what they are looking for, not where to find it. Their design consists of *Intentional Name Resolvers (INR)* that resolve intentional names and route messages. An application expresses the characteristics of the information or nodes they want to reach using *name-specifiers* in the message header instead of traditional source and destination addresses. When an INR receives a message, it performs the a lookup on the destination name-specifier in its name-trees (which stores the mapping between names and next-hop information) and forwards the message accordingly. This complements our work by providing the necessary functionality for implementing the service advertisement and discovery protocol.

Work on resource discovery includes distributed name server such as DNS and X.500 [8, 9, 17, 18]. More recently, there has been activity in the IETF to define protocol for locating services such as printers, disk server, etc., on local area networks. The Network Self managemenT & ORganization (NESTOR) project at Columbia University [10] is an example of a directory-based resource discovery project aimed for active networks. Success of this project would enable resource vendors and operations staff to code configuration information of arbitrary resources and use it to fully automate configuration changes and assure consistency and recoverability of network configurations.

Three programmable switch projects are being independently pursued by three research groups: MIT's ANTS [19], Georgia Tech's CANEs [20], and UPenn/Bellcore's SwitchWare [21]. Sortie takes the programmable switch concept to the next level of self-organizing networks. While programmable switch projects focus on solving the hard questions on how to support active services at a switch, Sortie focuses on how to design algorithms and protocols that allow network services to self-organize to improve the efficiency of routing and replica placement across the whole network.

Finally, Caching of Web objects such as `ftp` files and web pages [22–26] relates to our work in the sense that cached Web objects are dynamically replicated across the network to reduce both the network traffic and response time. The Harvest Project [23] organizes proxy server in a hierarchical fashion. When a server is queried for an object, it tries to satisfy the request locally. If that fails, it uses the Internet Cache Protocol (ICP) [27] to forward the request to its siblings and parent caches.

VI. Conclusion and Future Work

In this paper, we presented Sortie which provides a framework for self-organizing network services based on very simple rules independently executed by each switch. Sortie uses demand-triggered replication and considers the rate-transformation factor to improve the placement of services. To enhance the stability of the system, it complements its placement technique with an exponential back-off algorithm.

Our results show that Sortie offers a balance between all the performance metrics considered. It achieves higher placement efficiency while maintaining reasonable stability even at low resource constraints. While our simulation makes some simplifying assumptions, it does not compromise the general conclusion of our results. Monitoring the demand for services is proven to be very useful for achieving a distributed placement of services.

There are still several issues that warrant further investigation. We should extend the results to larger, hierarchical networks that subsume multiple ISPs. The initial distribution of home switches constitutes an interesting problem. We have ignored their effect on the amount of deviation from the original path. From our analysis, we saw that as replication delay is increased, the amount of deviation from the original path is also increased. Thus, poorly-placed home switches can have a large effect on network performance. Strategic placement of such switches is thus desired.

Our discussion in the paper has assumed that each packet requires only one service to be applied to it. It is not inconceivable that a packet may require more than one service. For example, packets carrying video data in a teleconference may need the services of a video transcoder as well as that of a multicast delivery router. From a routing efficiency point of view, it is undesirable if packets that require multiple services must trace back their paths to locate the next service. We will explore two techniques to deal with the *service composition* problem. The first is *service layering* in which services are classified into one of several layers that impose functional ordering between them. A multicast service, for example, would be classified to be executed last to minimize resource consumption. The second is *clustered routing* in which services of the same layer are combined into a logical "supernode" made up of several nodes in close proximity to each other that together provide the requested combination of services.

References

[1] E. Amir, S. McCanne, and R. Katz, "An active service framework and its application to real-time multimedia transcoding," *Proc. of ACM SIGCOMM '98*, Sep. 1998.

[2] I. Kouvelas, V. Hardman, and J. Crowcroft, "Network adaptive continuous-media applications through self organised transcoding," *Proc. of the Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video*, Jul. 98.

[3] S. Ratnasamy and S. McCanne, "Inference of multicast routing trees and bottleneck bandwidths using end-to-end measurements," *Proc. of IEEE INFOCOM*, 1999.

[4] A. Technologies, "Freeflow." url: http://www.akamai.com, 1998.

[5] S. Bhattacharjee *et al.*, "Application-layer anycasting," *Proc. of IEEE INFOCOM '97*, Apr. 1997.

[6] L. Zhang *et al.*, "Adaptive web caching: Towards a new global caching architecture," *Computer Networks and ISDN Systems*, Nov. 1998.

[7] W. Adjie-Winoto, E. Schwartz, and H. Balakrishnan, "An architecture for intentional name resolution and application-level routing," Feb. 1999.

[8] G. W. Neufeld, "Descriptive names in X.500," in *Proceedings of ACM SIGCOMM*, pp. 64–70, 1989.

[9] L. L. Peterson, "The profile naming service," *ACM Transactions on Computer Systems*, vol. 6, pp. 341–364, November 1988.

[10] Y. Yemini and S. Trito, "Nestor: Technologies and protocols for self-managed and self-organizing networks." url: http://www.cs.columbia.edu/dcc/nestor, 1998.

[11] A. Khanna and J. Zinky, "The revised arpanet routing metric," *Proc. of ACM SIGCOMM '89*, pp. 45–56, Sep. 1989.

[12] L. Zhang, S. Shenker, and D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," *Proc. of ACM SIGCOMM '86*, pp. 133–147, Sep. 1991.

[13] S. Floyd and V. Jacobson, "The synchronization of periodic routing messages," *ACM/IEEE Transactions on Networking*, vol. 2, no. 2, pp. 122–136, Apr. 1994.

[14] P. Francis *et al.*, "An architecture for a global internet host distance estimation service," *Proc. of IEEE INFOCOM '99*, Mar. 1999.

[15] C. Huitema *et al.*, "Project felix: Independent monitoring for network survivability." url: ftp://ftp.bellcore.com/pub/mwg/felix/, Sep. 1997.

[16] B. M. Waxman, "Routing of multipoint connections," *IEEE Journal on Selected Areas in Communications*, vol. 6, pp. 1617–1622, December 1988.

[17] P. Mockapetris and K. Dunlap, "Development of the domain name system," *Proc. of ACM SIGCOMM '88*, pp. 123–133, 1988.

[18] T. Brisco, "DNS support for load balancing," April 1995. RFC-1794.

[19] J. Guttag *et al.*, "From internet to activenet." url: http://www.sds.lcs.mit.edu/activeware/, 1998.

[20] E. Zegura, K. Calvert, and E. Trevena, "Canes: Composable active network elements." url: http://www.cc.gatech.edu/projects/canes/, 1998.

[21] W. Sincoskie *et al.*, "Accelerating network evolution with a software switch for active networks." url: http://www.cis.upenn.edu/ switchware, 1998.

[22] P. B. Danzig, R. S. Hall, and M. F. Schwartz, "A case for caching file objects inside internetworks," Tech. Rep. CU-CS-642-93, University of Colorado, Boulder, March 1993.

[23] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz, "The Harvest information discovery and access system," *Proceedings of the Second International World Wide Web Conference*, pp. 763–771, October 1994. Available from `ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Harvest.Conf.ps.Z`.

[24] S. Glassman, "A caching relay for the world wide web," *Computer Networks and ISDN Systems*, vol. 27, pp. 165–173, November 1994.

[25] S. Bhattacharjee, K. Calvert, and E. Zegura, "Self-organizing wide-area network caches," Tech. Rep. GIT-CC-97/31, Georgia Institute of Technology, 1997.

[26] L. Fan *et al.*, "Summary cache: A scalable wide-area web cache sharing protocol," *Proc. of ACM SIGCOMM '98*, pp. 254–265, Sep. 1998.

[27] D. Wessels and K. Claffy, "Internet cache protocol (icp), version 2," tech. rep., Internet Engineering Task Force, May 1997. draft-wessels-icp-v2-03.txt.