# Improving Scalability of Data-Centric Services using In-Network Traffic Inspection

Congchun He, Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{congchun, vijayk}@cs.nyu.edu

## Abstract

*A growing number of network services are being constructed using the XML Web Services architecture. By design, client interactions with such services are governed by open protocols such as UDDI, WSDL, and SOAP that permit programmatic discovery and functionality invocation. Additionally, for a large number of currently deployed services, it is possible to interpret service requests as structured accesses against a (physical or virtual) database. These two trends suggest the possibility of building network intermediaries that can inspect traffic flowing between clients and services, infer models for service access patterns, and potentially improve service scalability by taking actions such as replication, request redirection, or admission control. This paper reports on our experience designing and implementing such a network intermediary architecture. Experiments on an emulated WAN using synthetic workloads show that our approach achieves significant performance improvements in client-perceived response time.*

## 1 Introduction

Thanks in part to a cross-industry standardization effort, a growing number of network services are being built to support programmatic interaction using the XML Web Services platform. Illustrative examples include Amazon Web Services [1], the Google Web APIs service [2], and imagery services such as Microsoft's MapPoint [3], TerraServer [5] and SkyServer [4]. However, responses to requests for such services are usually dynamically generated and hence considered "uncacheable" by traditional caching architectures.

One way of remedying this situation is by developing alternative caching architectures, which improve scalability and performance using on-demand replication and request redirection strategies based on service usage locality information. Such architectures benefit from two characteristics shared by these example services.

First, to permit discovery and convenient integration with different kinds of clients across heterogeneous networks, services expose their functionality using standard XML-based specifications such as WSDL, SOAP, and UDDI [8]. A key consequence of this design is that network messages flowing between clients and services have a well-defined structure, one which is easily inferred from the WSDL specification of the service interfaces.

Second, many of the web services deployed today emphasize interactions that can be categorized as structured data retrieval. A large fraction of client traffic directed towards such services takes the following form: requests identify attributes of the items of interest (book title or ISBN, search keywords, map locations, etc.) and responses return information about the selected items (book information, list of matching pages, maps or other imagery, etc.). This behavior permits one to associate a semantic structure with such services: requests can be viewed as if they were accessing relations in a multi-attribute database. The database can be physical (as in our example services), or logical. A consequence of having such a structure is that one can associate the region of attribute values covered by a group of requests to the service internal data required for servicing this group.

Together, these two characteristics provide the context for an architecture where a distributed collection of network intermediaries, augmented with some service-specific knowledge, can (1) inspect traffic flowing between clients and services, (2) infer models for how a service is being accessed (specifically, if there are any locality patterns), and (3) potentially improve service scalability and client-perceived access times by taking actions such as replication, request redirection, or admission control. The first characteristic, use of standard XML-based protocols, enables the intermediaries to operate in a service-agnostic fashion. The second characteristic, the semantic structure of database access, provides the mechanism for detecting locality and reasoning about appropriate actions. The intended use of this architecture is as a service hosting platform, which improves client experience in the same way that a content-distribution network improves client-perceived latency for accessing static or streaming web content.

The effectiveness of such an architecture is of course determined by the extent to which usage locality is presented in real-world data-centric web services. In [14], we investigated request logs from two such sites, SkyServer [4] and TerraServer [5], to analyze service usage locality across several dimensions: *data space, network regions and timescales*. Our results validate the potential of the proposed architecture by showing that both workloads exhibit high degree of spatial and network locality over different time epoches. For example, in TerraServer's log, among all requests for dynamic content, 10% of clients contributed to $\sim$83.94% of the requests and 99.94% of requests hit on 10% of the regions in the data space. Similarly, in SkyServer's log, we found that 10% of clients contributed to $\sim$99.95% of the requests and 84.04% of requests hit on 30% of the regions in the data space. These results suggest that creating appropriate replicas of portions of the overall service database at a few locations in the network can yield considerable performance improvements.

This paper reports on our experience in designing and implementing a network intermediary architecture that automatically detects usage locality and creates service replicas to realize the above potential. The rest of the paper is organized as follows. Section 2 discusses related approaches. The details of our design, covering the functionality of an intermediary node, its interactions with other entities, and distributed algorithms for service replica placement are presented in Sections 3 and 4. We have built a prototype implementation of our proposed architecture using Microsoft's ASP.NET, and in Section 5, we report on its performance on an emulated WAN using synthetically generated workloads. We discuss outstanding issues in Section 6, and conclude.

## 2 Related Work

The main ideas behind our intermediary architecture are (1) the in-network inspection of service requests, and (2) use of the collected information for making service replication and request redirection decisions. Others have looked at similar ideas across four broad areas: network-layer congestion control, web caching and content distribution, edge deployment of network applications, and caching techniques for database-backed web applications.

Researchers in the networking community have proposed inspection of message traffic to cope with hotspot congestion along links. For example, work by Mahajan et al. [22] considers how individual routers can detect *aggregates* responsible for DoS attacks or flash crowds, and then request upstream routers to "push back" (throttle) these flows. Our approach benefits from the availability of application-level semantics associated with the requests, which enables requests to not only be throttled but also redirected to dynamically created replicas.

Web caching infrastructures and content distribution networks such as Akamai, as well as recently proposed peer-to-peer caching infrastructures such as Squirrel [16] and Coral [12] improve performance of web sites serving static content by caching (or hosting) this content closer to the network edge. A key observation is that such systems are effective primarily because intermediate nodes understand the structure of the request (i.e., that it is HTTP and refers to a particular URI), and because there is a well-established notion of the object referred to by this URI (the web page). Our approach builds on this observation by imposing a similar structure for general web service requests and making explicit what data a request refers to, in order to cache these requests which are usually considered "uncacheable" due to their dynamically generated responses.

Within the web caching context, researchers have also looked at optimal placement of replicas [25] and coordinated page placement and replacement policies [18]. In [25], the authors work with a "complete" replication model, where all of the web server contents are available at each replica. Under the assumption that a client uses a single replica, the authors reduce the placement problem to a K-median problem with an objective of minimizing global access costs. The work in [18] on the other hand, assumes a "partial" replication model, where individual objects are placed onto nodes organized in a hierarchical *cluster-tree* topology. A minimal cost flow formulation and its greedy and amortized approximations, are used to determine which subset of the objects to place on which nodes so as to minimize global access costs. The placement problem we describe in Section 4 is somewhat different from but combines elements of both of these works. Like [18], we are interested in a partial replication model but differ in the fact that given the large volumes of data associated with the services of interest, the cost of (even partial) replica creation and maintenance cannot be ignored. [25] indirectly considers this cost (in limiting the number of replicas that are created) but our problem context needs it to be more directly accounted for.

Recent efforts have also cached part of the application functionality (e.g., servlet execution for dynamic web content) closer to the network edge. Akamai's EdgeSuite [6] and IBM's WebSphere [15] contain similar support, as do research projects such as Gemini [24] and Active Cache [9] that attempt to improve generation and delivery of dynamic web content. Our approach similarly attempts to move re-

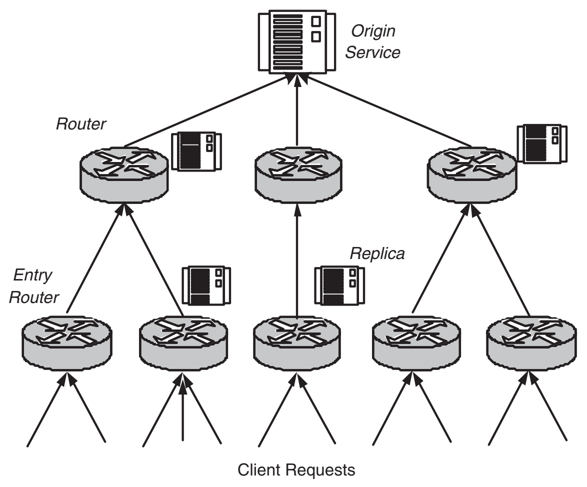


**Figure 1. Overview of the intermediary architecture (routers in the architecture are application-level routers)**

quest processing closer to clients, but differs in trying to determine automatically what this functionality should be. For achieving the latter it assumes that the functionality of a service replica can be implicitly specified in terms of regions of the physical or virtual database underlying the service.

From that perspective, more closely related are approaches that cache portions of the physical back-end database backing a web server on a local system to improve performance seen by queries originating from remote sites [11, 21, 7]. Of the different cache storage organizations that have been proposed, the work in this paper is most influenced by the notion of semantic regions [10]. Semantic regions refer to the range of relation values accessed by queries (requests). The differences in our approach are that (1) such regions may only be *virtual*, serving to specify the internal service data required for servicing a group of requests, (2) that these regions can dynamically split/collapse based on current load of requests, and (3) that the regions are cached across a distributed intermediary network instead of only at the endpoints.

## 3 Intermediary Architecture

Figure 1 shows an overview of our network intermediary architecture, which consists of service-neutral “router” nodes that interact with one or more service replicas. One of these replicas is assumed to always be active, and corresponds to the origin service. End clients connect to distinguished routers called *entry routers*. The router nodes are hierarchically organized, and relay requests and responses between the end-clients and service replicas[1].

In our architecture, the router network and the replica network can be maintained separately. Specifically, we assume that the service replicas are maintained by service providers; this permits service providers to offload service functionality and portions of the associated data on-demand onto the replicas without security concerns. The portions of data being offloaded onto or removed from a specific replica are determined by our distributed service replication algorithm and replication management mechanism, both of which run on the router network.

### 3.1 Router Functionality

Each of our routers is an application-level SOAP router, as defined by the Web Services architecture. The additional functionality they provide is the inspection of SOAP request messages to build a model for service usage; and the use of this model to improve service scalability and performance.

As stated earlier, to build this model, our architecture assumes a semantic structure for the service, namely that its requests can be treated as accessing relations in a logical multi-attribute data space. Each router keeps track of how portions of this data space are being accessed, and maintains metrics that summarize the performance of requests that access different portions. Based on this model of service usage, the routers cooperate with each other to determine what actions, if any, need be taken to improve service scalability and end-client performance. Actions supported by the architecture include requesting (from the replica network) activation of a service replica to service requests targeting a specific portion of the service’s data space, followed by redirection of affected request traffic to that replica. Section 4 discusses the specific algorithm we use to determine service action; here, we describe in more detail the request inspection and model construction.

To support the functionality described above, the router needs to (1) associate a SOAP request with the underlying logical data space for that service; (2) translate the parameters of the request into a region in this data space; (3) efficiently maintain usage statistics for that region; and (4) relay the request either to an upstream router or a nearby replica.

#### 3.1.1 Service Registration

The first two issues require active involvement of the service provider. We envision our architecture being used as

[1]To improve fault-tolerance, a node can be extended to a “super-node” that consists of a cluster of routers, which share information about usage statistics and serve requests in a load-balanced fashion. In this paper, we focus on the simple tree-structure when describing router functionality and interactions.

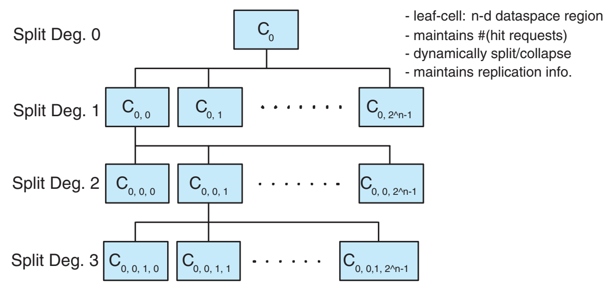


**Figure 2. Cell Structure**

a distributed hosting platform: services register with the architecture, and as part of the registration step supply the required information, which is then made available to all routers. Such information includes service interfaces (e.g., the SOAPAction attribute of a SOAP request to the service), the underlying logical data space for the service, the mapping scheme that maps a service request into a region in the data space, and the desired performance metrics (we focus on service response time in this paper). This step also identifies the entry routers, who publish, using UDDI or a similar protocol, their ability to receive service requests.

The logical data space is specified in a straightforward fashion, in terms of its dimensionality, the attributes corresponding to these dimensions, and the value ranges taken by these attributes. To translate arbitrary service requests into regions of this data space, we adopt a solution that leverages the XML-based nature of SOAP messages: the service specifies XSL stylesheets that are used by XSLT to transform each service request (identified by the corresponding SOAP action). This solution has the advantages of alleviating security concerns because XSLT is a safe language (modulo third-party extensions, whose use can be controlled), and of supporting the transformation in a language and platform-neutral fashion.

#### 3.1.2 Cell Structure

The third issue above, efficiently maintaining usage statistics for a data region, is handled traditionally by keeping track of the most popular requests and their responses in a cache-like structure. However, such an approach is not suitable in a network services context because (1) it is unlikely that a response to a previous request is reusable due to the variation of values and forms of request parameters; and (2) simple record keeping means that the router needs to store a high volume of requests (could be millions per day for TerraServer service), which leads to storage capacity concerns and results in inefficiency in analyzing the statistics at runtime. To solve these problems, we design a dynamic data structure called the *cell* structure, shown in Figure 2

Informally, a "cell" is a representation of a hyper-region in the logical service's data space, assuming that this data space is multi-attributed, and that the attributes are numerical or alphabetical rangeable. A cell maintains usage statistics about the corresponding region of the service's data space: these include the number of requests hitting the region over a time period, and the average service time seen by these requests (at a particular router).

When the number of requests hitting a cell exceeds a threshold, the cell can be *split* into a set of disjoint sub-cells, each of which covers a smaller region of the data space. Subsequent statistics are only maintained at the level of the sub-cells, each of which starts off with an equal share of the parent cell's hit count. Similarly, when the hit count of all sub-cells drops below a threshold, the sub-cells can be *collapsed* back into the parent cell. **Thus, at any point in time, a router maintains a cell tree, whose leaf nodes divide the service's data space into a set of disjoint regions**. The split and collapse operations permit efficient maintenance of statistics for different locality patterns involving coarse as well as fine-granularity regions.

Finally, the handling of the fourth issue, relaying of requests, also involves the cell structure. Information about regions of the service's data space for which nearby replicas are available are maintained as part of the cell structure. If no replica for that cell is available, the request is forwarded to the upstream router in the hierarchy.

### 3.2 Router-Router Interactions

In addition to cooperating to implement the service replication algorithm described in Section 4, the routers interact with each other to help compute the performance statistics seen by a group of requests. The downstream router (closer to the client) records the send time for every request forwarded to an upstream router. Upon receiving the response for this request, the router computes the *request service time*, which is defined to be the cumulative time spent by the request in traversing any routers upstream of this one and in processing at the service replica that satisfies the request. Each router, other than the entry router, piggybacks its own service time in the response sent to the downstream router, allowing the latter to maintain a dynamic estimate of the *round-trip time* being seen between the pair of routers for servicing a certain kind of request.

The measured request service time is used to update the cell statistics, and the round-trip time updates per-link statistics maintained at each router. Both pieces of information are used by the service replication algorithm.

### 3.3 Router-Service Interactions

After registration, the only direct interaction between our routers and the services whose requests they forward is for initiating replication actions. Depending on the outcome of the replication algorithm, a router may request that a replica be created to service requests targeting a certain region of the service's data space that is seeing unsatisfactory performance.

A replication request is sent to the origin service, which responds with a grant message in case it can create a replica near the requesting router. A subsequent confirmation message indicates the completion of the replication process, and results in an update to the cell structure with the replica information. The router can optionally queue up requests that it sees for the region being replicated in the interim period between a grant and its confirmation to allow the upstream network bandwidth to be utilized by the replica creation process. Note that the semantics of how the replica is created, and how the replicas are kept consistent with each other is left entirely up to the service. The network intermediary infrastructure merely identifies data space regions that ought to be replicated and the network regions that can benefit most from such replicas.

The router that requested the replication can also suggest that the replica is no longer required. This happens whenever the corresponding cell shows the region being accessed infrequently. To ensure behavior that is robust against temporary fluctuations, we adopt a policy that gradually ramps down a replication indicator field as long as the cell sees fewer than a threshold number of hits: when the field value reaches 0, the replica can be removed.

```
Inputs: (maintained per-router for each leaf-level cell)
    $\bar{T}^{\text{service}}$: average service time
    $t_{\text{lat}}$: round-trip time between router and parent
    $Q$: client-perceived response time threshold
        (approximated by service time at entry router)

Entry router:
    set $t_{\text{dec}} = \bar{T}^{\text{service}} - Q$
    if $t_{\text{dec}} < 0$
        send "Satisfied" message to parent
    else if $t_{\text{lat}} > t_{\text{dec}}$
        request replication
        send "Satisfied" message to parent
    else
        send ["Unsatisfied",$t_{\text{dec}}$] message to parent

Intermediate router:
    collect messages from children
    if all "Satisfactory"
        send "Satisfied" message to parent
    else
        set $t_{\text{dec}}$ = minimum $t_{\text{dec}}$ of children
        follow steps taken by Entry router

Root router:
    collect messages from children
    if any "Unsatisfactory"
        request replication
```

**Figure 3. Distributed algorithm for replica creation.**

## 4 Service Replication Algorithm

The replica placement problem has been well studied and shown to be an NP-complete problem for general network graph topologies. Systems have traditionally employed relatively simple heuristics such as demand-driven caching of frequently accessed (usually all) data at the network edge. More advanced approaches have also included some reasoning of data access patterns across multiple clients to determine where to place a replica. Example approaches in this category include the "best-client", "cascading replication" and "fast spread" mechanisms discussed in [26], which locate new replicas near clients that generate the most traffic, near other related replicas or along shared paths from clients to the origin service. In addition to such best-effort mechanisms, several researchers have also looked at formulations of the replica placement problem where the goal is to optimize some global metric, usually average client access costs. A representative formulation models the placement problem of placing a $M$ proxies on $N$ nodes as a K-Median problem [23]: for tree topologies, the latter problem admits an optimal solution based on dynamic programming, albeit with high complexity ($O(N^3M^2)$) [20], but approximations need to be employed for more general topologies [27, 25]. Researchers have also examined optimal strategies for the partial replication problem, when one needs to determine both the subset of replica objects and their placement [19, 18]. Most known results in this category have restricted themselves to hierarchical network topologies.

Our intermediary architecture described in the previous section can react to unsatisfactory performance by employing any of a number of algorithms, including the ones mentioned above. However, our problem context of data-centric network services precludes most of these algorithms from being directly applicable. First, given the volumes of data that such services involve, it may not be sufficient to cache accessed data only at storage-constrained edge servers. Second, and perhaps more importantly, even in situations where only a subset of the service data is being replicated, the costs of replica creation and maintenance cannot be totally ignored. Thus, our placement problem is closer to the optimal placement formulations described above and can be stated in a general form as follows: given information about client access patterns to different regions of the service's data space, determine what subset of the regions to replicate and at what locations in the network, so as to satisfy, with minimal cost of replica creation, client quality expectations on average response time (say, specified in terms of a maximum acceptable value).

For practical reasons, one would prefer algorithms that address this problem to enable a distributed implementation with minimal interaction between the participating routers. We describe one such distributed algorithm below, which, in the context of the hierarchical network topology we work with, attempts to optimize replica creation costs but for now ignores storage capacity constraints.

Although our algorithm employs well-known techniques, some of our design choices may be of wider interest. We assume a fairly general cost function to capture a variety of service-specific usage scenarios, requiring only that it be monotonic non-decreasing in the size of the replicated data space region and the distance of the replica location from the origin server.[2] Inputs to the algorithm include the cell-level usage statistics on data space regions collected by our routers, the measured service times, and the estimated round-trip times seen by requests between a router and its parent.

What makes the algorithm challenging is the cost aspect. If the latter were not a concern, one could simply replicate the affected regions of the data space (which are seeing unsatisfactory performance) on the routers closest to end clients. Factoring in the cost, it turns out that the replication problem as stated above is NP-hard, even on a simple chain-based router hierarchy because of interactions between different regions of the database. Consequently, we approximate a solution to the general problem by placing the restriction that the replication solution should satisfy the client response time threshold *independently* for each of the data space regions (leaf-level cells). This restriction breaks up the problem into a set of independent subproblems, each of which can be solved in polynomial time for tree-structured intermediary networks.

Figure 3 shows our distributed algorithm for the subproblem, which for simplicity is described in terms of the actions taken by routers in a particular round of the protocol. The basic idea is to use the round-trip service time estimates available at each router to determine if creating an upstream replica for the data space region can in fact satisfy the client-perceived response time expectations. The first router in the path from clients to the origin service that determines this

[2]The function can also incorporate any recurring consistency-maintenance costs.

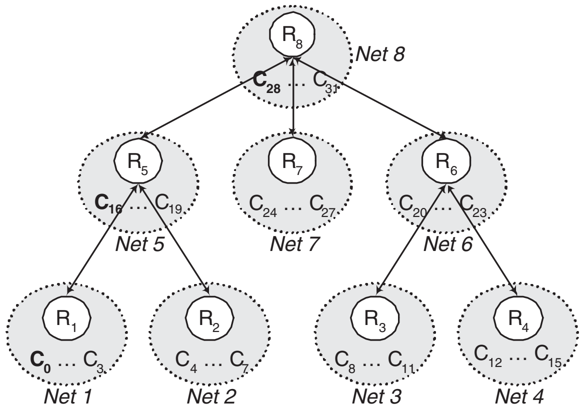


**Figure 4. Network configuration used in the experiments.**

in the negative is the one that ends up requesting the replica creation at a nearby site. The algorithm requires at most $D$ rounds, where $D$ is the depth of the hierarchy; the number of messages sent by a hierarchy involving $N$ routers is at most $2N$.

**Implementation notes** While we distinguish between entry, intermediate, and root routers in the hierarchy, note that a single router can play multiple roles in which case it simply performs all relevant actions. A practical implementation of the algorithm would set a threshold on the request hit-count seen by a cell before requesting replication; similarly, a threshold on the granularity of a cell could be set to prevent a large data space region from being replicated. Moreover, routers need not explicitly send satisfactory messages: the absence of such a message over a time window is assumed to indicate that the downstream router does not need any action. Finally, a service replication request need not always be sent to the origin service: an intermediate router can short-circuit the request to a nearby replica.

## 5 Evaluation

Our experiments use a synthetic service that reflects characteristics common to data-intensive web services such as MapPoint, SkyServer, or TerraServer. In defining the service data space and other request parameters, we used as a guide the real MapPoint service: our service supports queries for maps in the North America and the logical data space is defined by two attributes, latitude and longitude, and represents North America region at a resolution of 50000:1. The cell structure supports splitting up to 6 levels, partitioning the data space into a maximum of $2^{12}$ regions. The smallest region corresponds to map information at the

**Table 1. Network metrics on PlanetLab and our Click-based emulated WAN.**

| *PlanetLab* | | | | *Emulated Network* | | | |
|---|---|---|---|---|---|---|---|
| Path | RTT (ms) | b/w (Mb/s) | | Link | RTT (ms) | b/w (Mb/s) | |
| | | UDP | TCP | | | UDP | TCP |
| umich - caltech | 72.75 | 38.24 | 6.79 | net1 - net5 | 72.75 | 12.76 | 2.28 |
| umich - washington | 66.49 | 44.68 | 6.0 | net2 - net5 | 66.49 | 14.89 | 2.0 |
| columbia - cmu | 73.79 | 5.68 | 5.36 | net3 - net6 | 73.78 | 1.90 | 1.5 |
| columbia - princeton | 13.89 | 45.76 | 17.28 | net4 - net6 | 13.89 | 15.29 | 5.77 |
| nyu - umich | 46.25 | 37.84 | 7.52 | net5 - net8 | 46.25 | 12.62 | 2.51 |
| nyu - columbia | 9.31 | 45.63 | 10.34 | net6, 7 - net8 | 9.31 | 15.21 | 3.45 |

city level. Sizes of requests and responses also come from measurements against the real service: requests are 4 KB in length and responses are 34 KB. Consequently, replicating the smallest region requires transferring $\sim$11MB of data.

The experiments used a router implementation built on top of Microsoft's ASP.NET Framework 1.1. Each router was configured to recompute cell-based statistics every 300 seconds, and request replication only for cells that (1) received more than 500 requests over a period of 300 seconds; (2) were at a splitting level deeper than 3 (roughly corresponding to map information at the state level or smaller).

**Network configuration** Figure 4 shows an overview of the network configuration we use for the experiments. The configuration consists of eight network domains, each with a router and four clients that generate the service requests. The 8 router nodes ($R_1 \dots R_8$) are organized into a tree as shown in the figure, and serve as entry routers for the four clients in the same domain. $R_8$ acts both as a root router and as the location of the origin service. A service replica can be created on any of the other routers. We realized this configuration on a LAN cluster, emulating a WAN environment using the Click modular router infrastructure [17]. The specific emulation parameters came from measurements we took between pairs of PlanetLab hosts over an extended period (the bandwidth values were scaled down by a factor of 3 so as to accommodate the hardware limitation of a 100 Mb/s switch in our emulated system). Table 1 shows the close correspondence between the metrics measured on the two systems.

**Workload** Our clients repeatedly sent requests to the service, waiting for a response before sending the next request. To prevent saturating our underlying emulation system, each client was restricted to generating at most 5 requests every second (the actual rate may be lower due to congestion).

The workload generated by the clients reflects the results from [14], which showed that real workloads exhibit locality in both the regions of the service's data space they access and at the network level. To understand how our architecture and algorithms behave for these kinds of locality, our clients send requests according to the following pattern: they first select a rectangular region in the dataspace (*global region*) such that all clients within the same domain first agree upon a group center point within the global region, and then randomly request a point within a new rectangular region (*domain region*) surrounding the group center point. Thus, by controlling how close the group centers of different domains are to each other (denoted as the parameter $\alpha$, whose value is the ratio of a side of the global region to the range of the corresponding dimension in the origin dataspace), and how large the rectangular region is for each group (denoted as the parameter $\beta$, defined similar to $\alpha$ above but for the domain region), we can generate workloads that exhibit either spatial locality, or network locality, or both.

For each of the experiments, all 32 clients generate requests against the service simultaneously. Our service replication algorithm was given a maximum client response time threshold of 500 ms as its input, and computed the usage statistics and round-trip time estimates dynamically as discussed in Section 3. The cost function we chose for replicating a region was a linear function of two parameters: volume of data to transfer and hop distance between the replica and the origin service. Moreover, to prevent overwhelming the network, we imposed the restriction that concurrent replica creations at a router had to happen in sequence.

The graphs presented below show the moving average of the response time observed by the last 20 requests received at a client, and is computed every second.

## 5.1 Results

We show and discuss only results for representative clients along the longest path in our network configuration: net1 ($c_0$) - net5 ($c_{16}$) - net8 ($c_{28}$). Since the network configuration and client behavior is symmetric, the performance

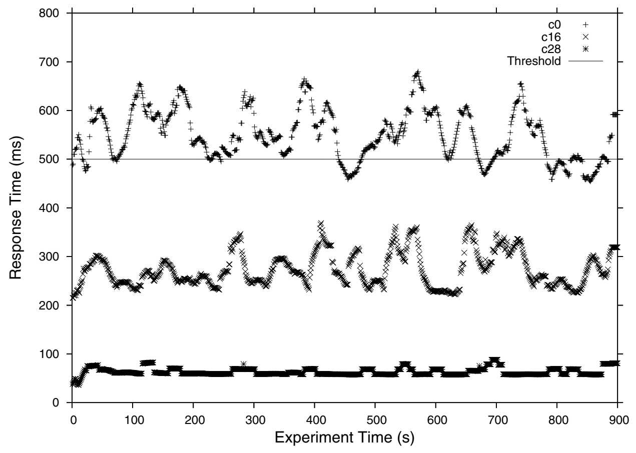


**Figure 5. Performance on an unloaded network.**

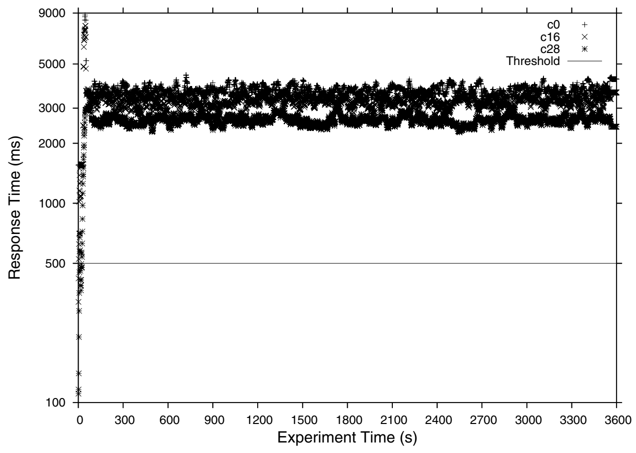


**Figure 6. Performance seen for a workload that exhibits both low spatial locality and low network locality ($\alpha = 1, \beta = 0.5$). Notice that the Y-axis is in logscale.**

of the other clients tracks the ones reported.

Figure 5 shows the baseline response time seen by clients in an unloaded network. The fluctuation in the response times stems from the (emulated) behavior of the network paths and typifies the same characteristics as our PlanetLab measurements. The response time has two components: round-trip time of the network path and the (non-overlapped) latency introduced by the computation at each router. The first component dominates: the computation at each router adds ~20 ms to the overall response time, with only 5 ms attributable to our code (the rest is caused by the .NET framework implementation).

Note that the response time threshold is not satisfied at $c_0$ even in the unloaded network. Once the network is loaded (Figure 6), without service replication, none of the clients satisfy the threshold (even clients in the same domain as the origin service due to the fact that the service needs to handle a large number of requests).

In the rest of this section, we focus on discussion of two interesting scenarios: a workload with low network locality but high spatial locality, and a workload with both high network locality and high spatial locality.

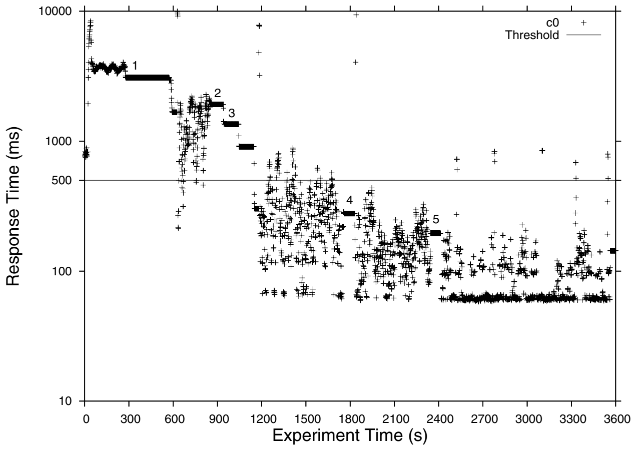
**Workload with low network locality, high spatial locality** ($\alpha = 0.5, \beta = 0.01$) Figure 7 shows that our infrastructure can dynamically detect the locality present within a domain and replicate portions of the service properly to satisfy client thresholds on response time. Given the locality structure, service regions are replicated at the router node in a domain to satisfy that domain's clients. In this case, 3 regions — 0030223,[3] 0030222, and 0032000 — with high access rates are replicated on $R_1$ starting at time 300 s, 900 s and 2400 s, respectively. Similarly, the 3 regions accessed by the clients in Net5 — 0212333, 0213220, and 0212331 — are replicated at the domain router, $R_5$, starting at time 300 s, 1200 s, and 2100 s, respectively. Note that each replication request results in a 11 MB data transfer across a congested network path: each such transfer takes approximately 300 s, and has the effect of serializing the replication requests from different routers. Consequently, it is only at time 2100 s and 2400 s that the clients of Net5 and Net1 see response times below their requested thresholds. The spikes in response times seen at various points in the graphs (e.g., at 600 s and 1200 s in the $c_0$ graph) can be explained as follows. Since a router queues up client requests for a region that is being replicated (the intuition here was to not have new requests compete with replica creation traffic for scarce network bandwidth), once the replication completes and these requests are serviced, their response times reflect the queuing time as well.

Figure 7 also shows two other interesting points. First, note that some regions were replicated in a redundant fashion: regions 0030222 and 0032000 are first replicated on $R_5$ and then re-replicated on $R_1$. There are two explanations for this, which point out the need for some refinements in our architecture:

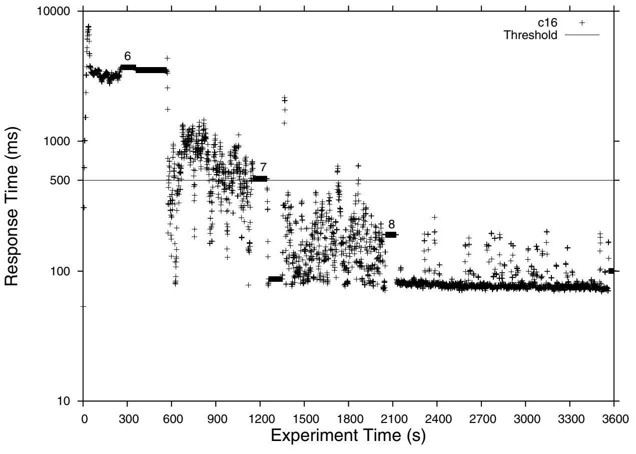
- Our replication algorithm looks at the current round-trip time estimate between an intermediary router and its parent to decide where best to perform the replication. However, once a replica is created at the parent,

[3]The region ID corresponds to the path in the cell tree taken to reach this region. For a 2-dimensional space, each split produces four subcells that are labeled 0–3. The $i^{th}$ digit in the region ID corresponds to the parent subcell of the current region at level $i$.

| Event | Region | Router | Replica lifetime | Event | Region | Router | Replica lifetime |
|---|---|---|---|---|---|---|---|
| 1 | 0030223 | $R_1$ | [300, -] | 6 | 0212333 | $R_5$ | [300, -] |
| 2 | 0030222 | $R_5$ | [895, 1620] | 7 | 0213220 | $R_5$ | [1200, -] |
| 3 | 0030222 | $R_1$ | [900, -] | 8 | 0212331 | $R_5$ | [2100, -] |
| 4 | 0032000 | $R_5$ | [1800, 2640] | | | | |
| 5 | 0032000 | $R_1$ | [2400, -] | | | | |



(a) $c_0$



(b) $c_{16}$

**Figure 7. Performance seen for a workload that exhibits low network locality but high spatial locality.**

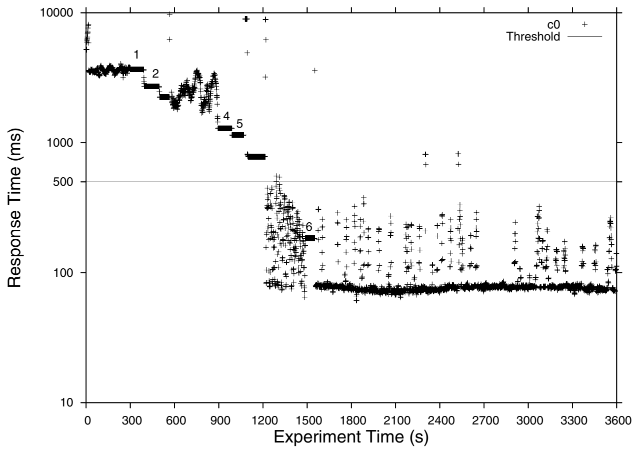
it is possible that the parent can service more client requests per unit time (remember that each client was configured to send a maximum of 5 requests per second, but the actual rate was $\sim$1.25 in a loaded network without any replication (Figure 6)). After replication, the increased incoming request rate at the parent increases queueing delays and hence the round-trip times seen by requests coming from the child router. What is required is a better way of estimating the round-trip time that would result after replication. Region 0032000 falls into this category.

- Each router operates asynchronously, with a thread waking up every 300 s to participate in the distributed replication algorithm. The following situation is thus possible: in one round, a router may find the request load for a region to be below the threshold required to request replication and consequently send an "Unsatisfied" message to its parent, while in the next round, the threshold may get crossed causing the router to initiate replication on its own. If the parent processes the first message during this period, it may end up seeing a request load that exceeds the configured threshold, and thus request replica creation on its own. In our experiment, region 0030222 falls into this category. Better synchronization between the routers would fix this problem.

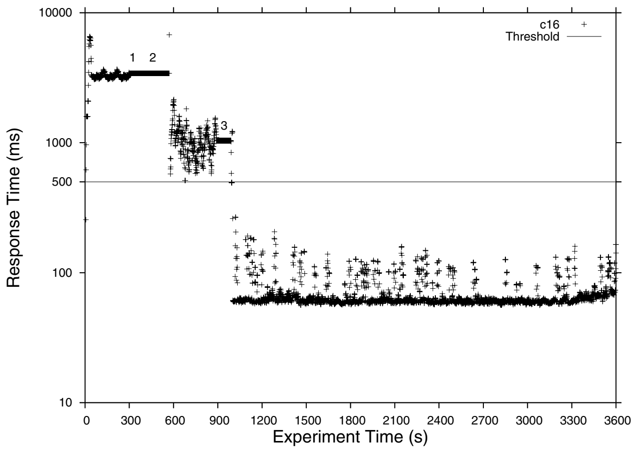
Note that both of these missteps are corrected in subsequent timesteps, with the replicas at $R_5$ getting reclaimed at 1620 s and 2640 s because of inadequate use. What is interesting is that before the replicas get reclaimed, they have an unexpected benefit: reducing the latency for the replication request for region 0030222 from region $R_1$ at time 2400 s, which is now satisfied by $R_5$ instead of going all the way to the origin service. This short-circuit manifests itself in the fact that response time seen by $c_0$ improves fairly quickly after the replication is requested, unlike the behavior observed for the earlier requests.

**Workload with high network locality, high spatial locality** ($\alpha = 0.01, \beta = 0.01$) Figure 8 shows that our infrastructure can successfully detect this kind of locality in the higher levels of the router hierarchy, and replicate service regions properly: regions 0211111 and 0122222 are the commonly requested regions and hence are replicated at both $R_1$ and $R_5$. In this case, the redundant replication is warranted: clients in Net5 need to have the region replicated in $R_5$ to satisfy their response time threshold requirement, while clients in Net1 cannot have their response time requirements satisfied with a replica at $R_5$ and hence, need a closer replica. Rerunning the experiment with the response time threshold raised to a higher value, 1500 ms, highlights this point: in this case, replicas at $R_5$ suffice for clients in

| Event | Region | Router | Replica lifetime | Event | Region | Router | Replica lifetime |
|---|---|---|---|---|---|---|---|
| 1 | 0211111 | $R_5$ | [300, -] | 4 | 0033333 | $R_1$ | [900, -] |
| 2 | 0122222 | $R_5$ | [300, -] | 5 | 0211111 | $R_1$ | [900, -] |
| 3 | 0300000 | $R_5$ | [900, -] | 6 | 0122222 | $R_1$ | [1500, -] |



(a) $c_0$



(b) $c_{16}$

**Figure 8. Performance seen for a workload that exhibits both high network locality and high spatial locality.**

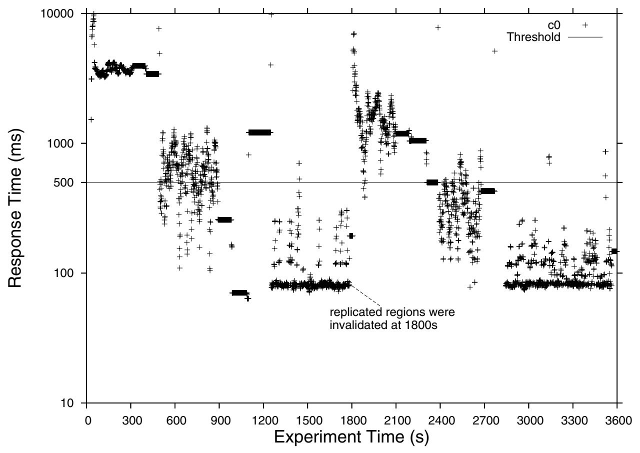


**Figure 9. Performance in the presence of a data update for a workload that exhibits both high network locality and high spatial locality.**

both Net5 and Net1.

**Workload with data update** ($\alpha = 0.01, \beta = 0.01$) Figure 9 shows the dynamic behavior of the infrastructure. A data update event on the origin service at time 1800 s causes the service replicas to be invalidated, and consequently re-acquired as needed. The response time observed by client $c_0$ stays within the desired threshold in the time period from 1200 s to 1800 s after the required regions were replicated on $R_1$. The data update event at 1800 s causes these regions to be invalidated, and results in a sharp increase of response time observed by the clients. The infrastructure reacts to this change by determining that these regions do need to be replicated again, which happens at 2300 s and 2600 s, resulting in reduced response time again from 2700 s onwards.

## 6 Discussion

Our architecture and algorithms have been developed and evaluated in the context of data-intensive XML web services hosted on intermediaries organized into a tree topology. Here, we discuss some ways in which the infrastructure can be extended and refined to permit broader applicability of the underlying ideas. We are currently working on the design and implementation of a more general architecture that embodies these ideas.

**Applicability to general network services** The two central ideas of this paper — in-network traffic inspection to build a model of service usage, and using this model to suggest service reactions to improve client performance — are equally applicable to services that are not web services. XML web services do make it simple to distinguish request messages from other traffic on the wire, to extract certain parameters from the request messages, and to, in a service- and platform-neutral fashion, relate these parameters to a region in the service's logical data space. However, the only requirement is for an intermediary to be able to look at a message's contents and ascribe a semantic structure to them.

Our description of the architecture and algorithms integrated two functions: locality detection embodied in shape of the cell-tree, and service replication. These two can be decoupled. For example, one might prefer using the locality detection functionality alone to model service usage patterns and then use this information to determine a static data partitioning strategy. To support applications of this nature, our architecture would need to be extended to detect locality over several timescales, as opposed to the current implementation that focuses on short-term patterns.

Our assumption of a tree-based hierarchy for the SOAP routers was motivated by the observation that wide area networks lend themselves naturally to this structure. Our service replication algorithm makes use of this property to prove its optimality. In practice, the algorithm can be extended to work on a more general topology where the hierarchy is made up out of clusters of network intermediaries: routers within the same cluster share usage statistics with each other and coordinate their decision making.

**Exploiting additional service structure** Our architecture works with the notion of a logical "view" of a dataspace to model service usage, assuming that the details of the back-end database may either not exist or are unlikely to be exposed. In cases where the service owner would like to expose such information, the service replication algorithm can and perhaps should be extended to come up with a replication solution for regions of the backend database as opposed to the materialized view [13] embodied in the responses. The former is likely to result in a lower amount of redundancy as compared to the latter.

**End-to-end security** We have assumed a trust relationship between the service owner and the intermediary architecture. This manifests itself, among other places, in the fact that in permitting inspection of SOAP messages, we have assumed that messages are either not end-to-end encrypted, or when they are, the service permits their decryption at the intermediate sites. When the router is only partially trusted by the service, we can relax this assumption by requiring that only a portion of the message body be made public (similar to the notion of *message properties* in BPEL4WS). As long as these properties suffice to associate a request with the service's data space, the benefits of the infrastructure can be made available while still protecting sensitive information. The trust assumptions for relaying of requests and responses are no different from that made for network-level routers in current Internet-scale networks.

**Hosting multiple services** Although we have demonstrated how our architecture operate on a single Mappoint-like web service, the intended use of our architecture is as a service hosting platform. To accommodate competition for computation and network bandwidth among multiple services being hosted on a router, additional resource sharing policies are needed. Notice that different services need not share the same underlying hierarchical network topology: for each service, an intermediary separately maintains the information about its parent and the origin web server.

## 7 Summary

This paper has described a network intermediary architecture, which leverages in-network detection of locality patterns of web service usage to improve service scalability using replication, request redirection, or admission control. The architecture was motivated by our analyses of request logs from two production services, which showed that real workloads exhibit substantial locality across several dimensions. Our experience implementing this architecture and evaluating its performance on an emulated WAN against synthetic workloads shows that the approach has the potential of achieving significant performance improvements in client-perceived response time.

## Acknowledgment

This research was sponsored by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929; by NSF grants CAREER:CCR-9876128, CCR-9988176, and CCR-0312956; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

## References

[1] Amazon Web Service home page. `http://www.amazon.com/gp/aws/landing.html`.

[2] Google Web APIs Home. `http://www.google.com/apis/`.

[3] Microsoft MapPoint Web Service. `http://www.microsoft.com/mappoint/webservice/default.mspx`.

[4] Sloan Digital Sky Survey / SkyServer. `http://skyserver.sdss.org/`.

[5] TerraServer Web Services. `http://terraservice.net/webservices.aspx`.

[6] Akamai Technologies Inc. Edgesuite services. `http://www.akamai.com/html/en/sv/edgesuite_over.html`.

[7] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, March 2003.

[8] T. Berners-Lee. Web Services overview - Design Issues. *W3C Web Services Workshop*, July 2003.

[9] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proc. of IFIP Int'l Conf. Dist. Sys. Platforms and Open Dist. Processing*, 1998.

[10] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. 22nd VLDB Conference*, pages 330–341, 1996.

[11] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. On the placement of web server replicas. In *Middleware Conference*, pages 24–44, 2001.

[12] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *Proc. 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, March 2004.

[13] A. Gupta and E. Inderpal S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, November 1998.

[14] C. He and V. Karamcheti. An Analysis of Usage Locality for Data-Centric Web Services. Technical Report TR-2005-866, New York University, 2005.

[15] IBM Corp. Websphere platform. `http://www.ibm.com/websphere`.

[16] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. PODC*, July 2002.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[18] M. R. Korupolu and M. Dahlin. Coordinated placement and replacement for large-scale distributed caches. *IEEE Transactions on Knowledge and Data Engineering*, 14(6), Nov/Dec 2002.

[19] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. In *Proc. of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 586–595, 1999.

[20] B. Li, M. Golin, G. Italiano, and X. Deng. On the optimal placement of web proxies in the internet. In *Proc. of the IEEE INFOCOM'00*, March 2000.

[21] Q. Luo and J. F. Maughton. Form-based proxy caching for database-backed web sites. In *VLDB Conference*, pages 191–200, 2001.

[22] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *SIGCOMM Computer Communications Review*, 32(1), 2002.

[23] P. Mirchandani and R. Francis. *Discrete Location Theory*. John Wiley and Sons, 1990.

[24] A. Myers, J. Chuang, U. Hengartner, Y. Xie, W. Zhang, and H. Zhang. A secure and publisher-centric web caching infrastructure. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*, April 2001.

[25] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *Proc. of the IEEE INFOCOM'2001*, April 2001.

[26] K. Ranganathan and I. T. Foster. Identifying dynamic replication strategies for a high-performance data grid. In *Proc. of the 2nd Int'l Workshop on Grid Computing*, pages 75–86, November 2001.

[27] A. Vigneron, L. Gao, M. Golin, G. Italiano, and B. Li. An algorithm for finding a k-median in a directed tree. *Information Processing Letters*, 7:81–88, 2000.