Optimizing Data Aggregation for Cluster-based Internet Services

Lingkun Chu, Hong Tang, Tao Yang *
Dept. of Computer Science
University of California at Santa Barbara
{lkchu, htang, tyang}@cs.ucsb.edu

Kai Shen
Dept. of Computer Science
University of Rochester
kshen@cs.rochester.edu

ABSTRACT

Large-scale cluster-based Internet services often host partitioned datasets to provide incremental scalability. The aggregation of results produced from multiple partitions is a fundamental building block for the delivery of these services. This paper presents the design and implementation of a programming primitive – Data Aggregation Call (DAC) – to exploit partition parallelism for cluster-based Internet services. A DAC request specifies a local processing operator and a global reduction operator, and it aggregates the local processing results from participating nodes through the global reduction operator. Applications may allow a DAC request to return partial aggregation results as a tradeoff between quality and availability. Our architecture design aims at improving interactive responses with sustained throughput for typical cluster environments where platform heterogeneity and software/hardware failures are common. At the cluster level, our load-adaptive reduction tree construction algorithm balances processing and aggregation load across servers while exploiting partition parallelism. Inside each node, we employ an event-driven thread pool design that prevents slow nodes from adversely affecting system throughput under highly concurrent workload. We further devise a staged timeout scheme that eagerly prunes slow or unresponsive servers from the reduction tree to meet soft deadlines. We have used the DAC primitive to implement several applications: a search engine document retriever, a parallel protein sequence matcher, and an online parallel facial recognizer. Our experimental and simulation results validate the effectiveness of the proposed optimization techniques for (1) reducing response time, (2) improving throughput, and (3) gracefully handling server unresponsiveness. We also demonstrate the (4) ease-of-use of the DAC primitive and (5) the scalability of our architecture design.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms

Algorithms, Design, Experimentation, Performance, Reliability

Keywords

Scalable data aggregation, response time, throughput, load-adaptive tree formation, cluster-based network services, fault tolerance

1. INTRODUCTION

Computer clusters are widely deployed to deliver highly scalable and available online services [2, 6, 13, 15]. Well-known Web sites such as Yahoo and MSN employ service clusters with thousands of machines. The persistent data for cluster-based Internet services are often partitioned and the aggregation of data produced from multiple partitions is a commonly requested operation. For instance, an online discussion group service may partition data based on discussion topics. To serve a client request looking for articles posted by a particular author, the service cluster needs to perform searches on all data partitions and aggregate the results before replying back to the client. Similar examples that require parallel service invocation and result aggregation can be found in an online auction service where auction items may be partitioned based on categories; or in an Internet search engine where data may be partitioned based on their URL domains.

Supporting efficient data aggregation is not straightforward. Several previous research projects on cluster-based service programming rely on a fixed node to aggregate results [6, 19], which could quickly run into scalability problems when a large number of partitions are involved. On the other hand, it is desirable to provide a high-level data aggregation primitive to aid service programming and hide the implementation details behind an easy-to-use interface. A good implementation not only needs to optimize both response time and system throughput, it also needs to consider platform heterogeneity caused by hardware difference, varying network conditions, and non-uniform application data partitions. Furthermore, it needs to handle node failures and unresponsiveness caused by hardware faults, software bugs, and configuration errors.

This paper studies the programming support and architecture design of scalable data aggregation operations for cluster-based Internet services. We propose a service programming primitive called *Data Aggregation Call* or DAC to merge data from multiple partitions. Additionally, the DAC provides two options for online services to specify *soft deadline guarantee* and *aggregation quality guarantee* when a DAC invocation can return partially aggregated results [5]. The objective of our architecture design is to improve response time with sustained system throughput. Additionally, our

* Also affiliated with Ask Jeeves/Teoma.

PPoPP'03, June 11–13, 2003, San Diego, California, USA.

design targets large-scale service clusters where platform heterogeneity and node failures are common. In this paper, we propose three techniques to achieve our goal. (1) At the cluster level, we use a load-adaptive tree formation algorithm that balances load across servers. (2) Inside each cluster node, we use a highly concurrent event-driven request scheduling scheme that prevents slow responding nodes from blocking working threads and adversely affecting system throughput. (3) To avoid slow or failed nodes from delaying the completion of DAC requests, we introduce a staged timeout scheme that eagerly prunes out slow or failed servers from a reduction tree.

The work described in this paper is a critical building block in *Neptune*, a middleware system that provides replication support [19], and quality-aware resource management [17, 18] for scalable cluster-based network services. We have applied the DAC primitive in the implementation and deployment of several applications: a search engine document retriever, a parallel protein sequence matcher, and an online parallel facial recognizer.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the Neptune clustering middleware. Section 3 describes the semantics of the DAC primitive. Section 4 discusses our runtime support techniques for DAC. Section 5 presents the evaluation of the proposed DAC primitive and individual techniques used in the DAC implementation. Section 6 discusses related work and Section 7 concludes the paper.

2. NEPTUNE: CLUSTERING SUPPORT FOR SCALABLE INTERNET SERVICES

The work described in this paper is part of the *Neptune* framework – programming and runtime support for building cluster-based Internet services [19]. This section presents a brief background overview of the Neptune clustering architecture and its programming environment.

2.1 Clustering Architecture

Neptune targets cluster-based network services accessible to Internet users. Requests issued by remote clients enter service clusters through protocol gateways such as Web servers or XML gateways. Inside the service cluster, services are typically composed of several service components. Persistent data for service components are usually partitioned and replicated for incremental scalability and high availability. We use the term *Service Instance* to denote a server entity that runs on a cluster node and manages a data partition belonging to a service component. Neptune employs a functionally symmetric and decentralized clustering design. Each service instance can elect to provide services (when it is called *service provider*) and it can also acquire services exported by other service instances (when it is called *service consumer*). This model allows multi-tier or nested service architecture to be easily constructed.

Figure 1 illustrates the architecture of a prototype document search service supported by the Neptune middleware. In this example, the service cluster delivers a search service to Internet users and business partners through Web servers and XML gateways. Inside the cluster, the main search task is decomposed into two parts and distributed to index servers and document servers. The data for both components is partitioned and replicated. In Figure 1, there are two index server partitions and three document server partitions. Each partition has three replicas. The arcs labeled with ①–④ in Figure 1 show a simplified work flow of serving a client request. ① A search query arrives at one of the protocol gateways. ② The protocol gateway contacts the index server partitions to retrieve the identifications of documents relevant to the search query. ③ The protocol gateway contacts the document server partitions which

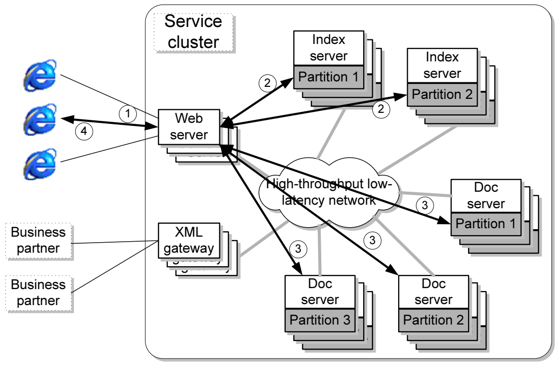

Figure 1: A prototype search engine service supported by Neptune

translate the list of document identifications to human understandable descriptions. ④ Finally, the protocol gateway compiles the final results in HTML or XML format and returns them back to the client. In the above work flow, the protocol gateway contacts the service instances through the *Neptune consumer module*. On the service provider side, the requests are received by the *Neptune provider module*, which subsequently invokes the service-specific handlers to process the requests.

Neptune provides many reusable functionalities in the middleware layer to ease the service construction. (1) **Naming, service location, and load balancing:** Service component partitions are addressed through location-transparent names (service name, partition ID). The Neptune consumer module automatically routes each request to an appropriate node based on the service availability and runtime workload. (2) **Replication consistency:** Neptune provides several levels of consistency guarantees for consumer requests involving data updates, and an application can choose the desired level based on its service semantics. (3) **Fault isolation and failure recovery:** The Neptune provider module at each node periodically announces a service availability, or heartbeat, message to other nodes. Faulty nodes will be automatically detected by the discontinuance of heartbeat messages. When a failed node recovers, Neptune automatically performs data consistency maintenance if necessary [19].

2.2 Neptune Programming Support

Neptune allows service programmers to concentrate on the functional design of service components without being concerned with the details of the underlying clustering architecture. The Neptune service call interface hides the complexity of request routing and network communication management behind a set of easy-to-use function call interfaces. We briefly discuss how to implement an online service in Neptune.

As mentioned before, each service provider exports certain functionalities to service consumers through request/response sessions. Inside each service provider, Neptune maintains a request queue and a thread pool to handle requests concurrently, and *application programmers only need to specify a set of service-specific request handlers*. These handlers are compiled into a dynamic library module and they are loaded into the Neptune runtime system when needed. The Neptune provider modules process requests by calling the corresponding request handlers.

When a service instance seeks certain functionality from another service instance, it uses the Neptune consumer module's service call interface to communicate with an appropriate service instance acting as a service provider. Neptune supports two styles of interactions between service consumers and service providers. In the *message*-based scheme, a service consumer specifies the request and response buffers in one function call, which bears some similarity with RPC (Remote Procedure Call). This scheme is suitable for interactions involving small amount of data. The *stream*-based scheme requires a service consumer to first establish a stream connection between the service consumer and provider, and then interacts with the service provider through the bidirectional channel. This scheme allows the exchange of large amount of data without pre-allocating buffering space.

A number of applications have been deployed on the Neptune platform, including an online auction service, an online forum service, and a persistent cache utility. Our experience with Neptune as a service programming platform has been very positive and most of these deployments were completed within a few days. In particular, Neptune has been successfully used as the clustering middleware at the Web search engine sites Teoma and Ask Jeeves since 2000. As of Fall 2002, the system grows to over 900 multiprocessor machines.

3. DAC SEMANTICS AND PROGRAMMING INTERFACE

In this section, we will first present the semantic design of the Data Aggregation Call (DAC) primitive, followed by a description of its programming interface. In the end, we will compare it with the MPI reduction primitive.

3.1 The Basic DAC Semantics

A generic data aggregation operation over a set of partitions can be viewed as a composition of two types of basic operations. First, a *local processing operation* (called a χ_{LOCAL} operator) is performed on every participating partition, which processes the dataset of that partition and produces an output dataset. Secondly, the output datasets produced from all participating partitions are aggregated into one output dataset through repeated invocations of a *global reduction operation* (called a χ_{REDUCE} operator). A χ_{REDUCE} operator takes in two source datasets and produces one output dataset. For example, in Figure 1, the retrieval of matching document identifications from all partitions (the arcs labeled with 2) is a data aggregation operation. The χ_{LOCAL} operator in this aggregation operation selects a list of document identifications related to a search query from a local partition. The χ_{REDUCE} operator sorts two lists of document identifications (based on their relevancy to the query) and merges them into one list. Notice that we use term *reduction* instead of *aggregation* when an operation is binary.

Figure 2 specifies the basic semantics of the DAC primitive through a sequential algorithm, which reflects the idea of the generic data aggregation operation described in the previous paragraph. Note that the sequential algorithm shown in Figure 2 is only meant to specify the desired outcome of a DAC invocation while a different algorithm (possibly a parallel algorithm) could be used for the actual implementation.

The two operators for the DAC primitive, χ_{LOCAL} and χ_{REDUCE}, deserve some further discussion. There is no formal restriction for χ_{LOCAL} and it can be any operations performed on a single partition. Typically, the χ_{LOCAL} operator involves the selection of a sub-dataset from a data partition followed by a transformation that produces the output data from the selected sub-dataset. On the other hand, we do assume the χ_{REDUCE} operator to be both commutative and associative, which is generally the case in practice [4, 11, 16, 20]. As will be shown in later sections, requiring the χ_{REDUCE} operator to be commutative and associative allows us to perform parallel reduction with limited synchronization.

Algorithm 3.1: DAC($P, \chi_{\text{LOCAL}}, \chi_{\text{REDUCE}}, I_{\text{LOCAL}}, I_{\text{REDUCE}}$)

Input: $P = \{p_1, p_2, ..., p_n\}$: The set of n participating partitions.
Input: χ_{LOCAL}: The local processing operator.
Input: χ_{REDUCE}: The global reduction operator.
Input: I_{LOCAL}: The input parameters for χ_{LOCAL}.
Input: I_{REDUCE}: The input parameters for χ_{REDUCE}.
Returns: r: The result dataset of the data aggregation call.

// First, we apply the local processing operation on all partitions.
for $i \leftarrow 1$ **to** n
  do $r_i \leftarrow \chi_{\text{LOCAL}}(p_i, I_{\text{LOCAL}})$

// Second, we aggregate the output datasets $\{r_1, r_2, ..., r_n\}$
// to the final result r through the global reduction operation.
$r \leftarrow r_1$
for $i \leftarrow 2$ **to** n
  do $r \leftarrow \chi_{\text{REDUCE}}(r, r_i, I_{\text{REDUCE}})$

return (r)

Figure 2: Specification of the basic DAC semantics.

3.2 Adding Quality Control to DAC

The specification in Figure 2 assumes that there is no cluster node failures and the request demand is below the system capacity. However, in a real operational environment, cluster nodes could be unavailable due to software or hardware problems. Additionally, client request demand level could fluctuate dramatically and it may exceed system capacity. It is critical to provide prompt responses for client requests under those situations.

The DAC primitive provides two additional input parameters to allow service programmers to control the behavior of a data aggregation operation in the event of node failures and system overload.

(1) **Aggregation quality guarantee:** For many online services, partial aggregation results may still be useful. We define the *quality* of a partial aggregation result as the percentage of partitions that have contributed to the partial result. Service programmers can specify an *aggregation quality guarantee* (a percentage threshold), and a partial aggregation result is only acceptable when its quality exceeds the threshold. For a service that cannot tolerate any partition loss in an aggregation operation, we can specify the threshold to be 100%. An aggregation quality guarantee below 100% allows the system to trade the quality of aggregation results for availability (i.e., the number of successfully fulfilled requests), which is important for large-scale network services [5].

(2) **Soft deadline guarantee:** Online service users typically demand interactive responses. As a result, we allow service programmers specify a service-specific *soft deadline guarantee* in a data aggregation call. The deadline guarantee provides guidance for the system to balance between the aggregation quality and the promptness of request responses. It also allows the system to eagerly abort requests that stand a low chance of meeting the deadline. This avoids wasting resources for serving these requests, which are likely to be discarded by online users anyway. The deadline guarantee is *soft* in the sense that the DAC may return with a response time slightly over the specified deadline.

The aggregation quality and soft deadline guarantees essentially make the semantics of the DAC primitive non-deterministic. Figure 3 shows the complete semantics of the DAC primitive, whose non-determinacy is manifested in two places – the two possible execution paths, and the non-deterministic subset of the contributing partitions.

Algorithm 3.2: DAC($P, \chi_{\text{LOCAL}}, \chi_{\text{REDUCE}}, I_{\text{LOCAL}}, I_{\text{REDUCE}}, \rho, T$)

Input: $P = \{p_1, p_2, ..., p_n\}$: The set of n participating partitions.
Input: χ_{LOCAL}: The local processing operator.
Input: χ_{REDUCE}: The global reduction operator.
Input: I_{LOCAL}: The input parameters for χ_{LOCAL}.
Input: I_{REDUCE}: The input parameters for χ_{REDUCE}.
Input: ρ: The aggregation quality guarantee ($0 < \rho \leq 1$).
Input: T: The soft deadline guarantee.
Returns: $\langle s, Q, r\rangle$: The status s (**success** or **fail**), contributing partitions Q, and aggregation result r.

Execution path 1.
// Request cannot be fulfilled due to node failures
// or resource constraint.
return ($\langle$ **fail** , **nil** , **nil** $\rangle$)

Execution path 2.
// Request fulfilled within or close to the deadline T.
$Q \leftarrow$ *a subset of P with m ($m \geq \rho \times n$) partitions*
for $i \leftarrow 1$ **to** m
  do $r_i \leftarrow \chi_{\text{LOCAL}}(q_i, I_{\text{LOCAL}})$
$r \leftarrow r_1$
for $i \leftarrow 2$ **to** m
  do $r \leftarrow \chi_{\text{REDUCE}}(r, r_i, I_{\text{REDUCE}})$
return ($\langle$ **success** , $Q, r\rangle$)

Figure 3: The complete DAC semantics. The non-determinacy of the semantics is manifested in two places. (1) the two execution paths; (2) the non-deterministic subset Q.

3.3 DAC Programming Interface

The DAC programming interface consists of two parts. On the service consumer side, it specifies how to invoke a DAC request. On the service provider side, it specifies how to write service handlers (callback functions) that will be used by the Neptune run-time system to fulfill DAC requests.

The C++ interface for DAC invocation is shown in Figure 4 (a variation for stream-based calls is not presented due to the space constraint). The first argument is an opaque Neptune client handle which links to the states of a Neptune client (service consumer). This handle is obtained during the instantiation of a Neptune client. The class `NeptuneCall` specifies a registered request handler on a Neptune service provider (such as the name and version of the handler). The class `NeptuneData` maintains structured and typed data in a platform-independent manner.

```
bool NeptuneDAC(
  // Input parameters:
  NeptuneCltHandle & h,             // Neptune client handle
  char *             svc_name,      // service name
  set<int> &         partitions,    // participating partitions
  NeptuneCall &      local,         // local processing operator
  NeptuneCall &      reduce,        // global reduction operator
  NeptuneData &      prm_local,     // parameters for local
  NeptuneData &      prm_reduce,    // parameters for reduce
  double             aqg,           // aggreg. quality guarantee
  double             deadline,      // soft deadline guarantee
  // Output parameters:
  NeptuneData &      result,        // aggregation results
  set<int> &         ctrb_parts     // contributing partitions
);
```

Figure 4: The C++ interface for the DAC primitive.

On the service provider side, a service library provides the implementation of two callback functions corresponding to the two operators specified in the DAC interface. The functions are required to take the **`typedef`** interfaces shown in Figure 5. For both interfaces, the first parameter is an opaque Neptune service handle which links to the states of a Neptune service instance. Note that the interfaces shown in Figure 5 pass input and output data in a message-based style through the `NeptuneData` objects.

```
// local processing operator interface definition
typedef bool op_local(
  // Input parameters:
  NeptuneSvcHandle & h,             // Neptune service handle
  int                part_id,       // partition ID
  NeptuneData &      parameters,    // request parameters
  // Output parameters:
  NeptuneData &      result         // local processing result
);

// global reduction operator interface definition
typedef bool op_reduce(
  // Input parameters:
  NeptuneSvcHandle & h,             // Neptune service handle
  NeptuneData &      src1,          // reduction input source 1
  NeptuneData &      src2,          // reduction input source 2
  NeptuneData &      parameters,    // request parameters
  // Output parameters:
  NeptuneData &      result         // reduction output
);
```

Figure 5: The C++ `typedef` interfaces for operator callback functions. A local processing operator is required to have the type `op_local`, and a global reduction operator is required to have the type `op_reduce`.

3.4 Comparing DAC with MPI Reduce

Conceptually, DAC is similar to the MPI *reduce* primitive which aggregates data from participating MPI nodes to a root node through a reduction operator. To a certain degree, the design of the DAC programming interface has been influenced by the MPI reduce primitive [8, 20]. However, our work differs from prior work for supporting MPI reduce primitives in significant ways. First, MPI reduce does not tolerate node failures, so the MPI reduce operation would fail if any of the participating MPI node fails. Second, MPI programs are less restrictive on interactive responses and MPI reduce does not require deadline guarantee. Finally, MPI relies on a procedure programming model while Neptune uses a stateless request-driven model. As a result, service programmers must specify an additional local processing operator in a DAC invocation.

4. RUNTIME SUPPORT

There are two objectives in the runtime system design for DAC. The first objective is to minimize the service response time with sustained throughput in both homogeneous and heterogeneous environments. Our second goal is to minimize the impact of node failures and unresponsiveness. Figure 6 shows the overall system architecture for DAC. Upon receiving a DAC invocation, the DAC client module constructs a reduction tree and assigns participating service instances to tree nodes. It then multicasts this information with the actual request to all participating providers. All service providers then perform the local processing operation, and cooperate together to aggregate data from the bottom of the tree to the root. Specifically, each provider aggregates the local processing results with datasets returned from its children. Once it finishes aggregating data from all its children (or if it does not have any child, or the request times out), it then sends the local aggregation results to its parent (or sends back the final result to the original service consumer if it is the root). This process continues until the final result reaches the service consumer.

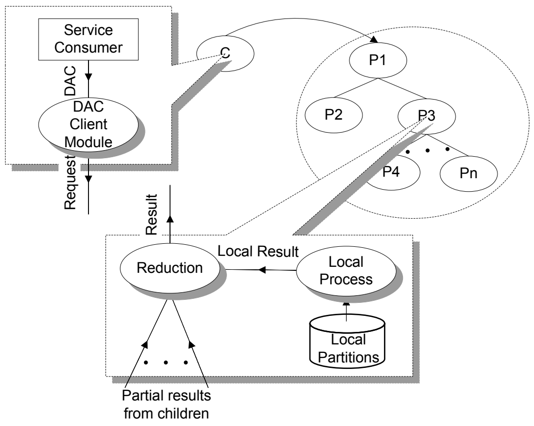

Figure 6: **Overall system architecture for DAC.**

On top of this architecture, our runtime system design targets two issues: (1) How to build a global reduction tree to exploit partition-level parallelism? (2) How to efficiently and reliably serve highly concurrent service requests on each provider? We address the first issue in Sections 4.1 and 4.2. The second issue will be studied in Sections 4.3 and 4.4. Section 4.5 discusses several other implementation issues.

4.1 Reduction Tree Formation Schemes

In this section, we present three reduction tree formation schemes, as illustrated in Figure 7.

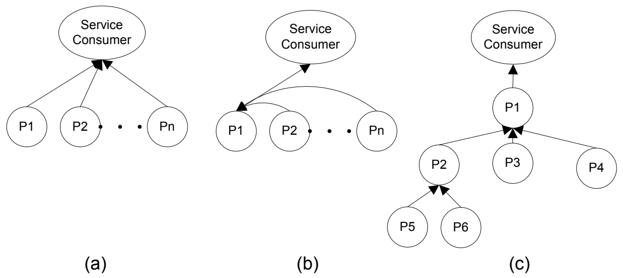

Figure 7: **Tree formation schemes: (a) Base (b) Flat (c) Hierarchical.**

(1) **The baseline scheme (Base):** The first scheme is the baseline scheme that performs data aggregation when there is no special support from the runtime system. As shown in Figure 7 (a), in the *Base* scheme, a service consumer initiates invocations on all participating partitions, collects results from these partitions, and aggregates these results. The main problem of this approach is that the service consumer is responsible for all aggregation work and may quickly become the bottleneck when request demand increases.

(2) **Flat tree with delegated roots (Flat):** In the second scheme, as shown in Figure 7 (b), a service consumer delegates the aggregation work to a service provider, and chooses different delegated roots for different DAC requests. This allows the system to handle high request demand without saturating either the service consumer or any of the service providers. However, the response time of this *Flat* scheme is not satisfactory because the aggregation work is serialized on a root node.

(3) **Hierarchical tree with delegated roots (Hierarchical):** As shown in Figure 7 (c), the third scheme improves from the Flat scheme by replacing the flat reduction tree with a hierarchical tree. A balanced hierarchical tree with appropriate width and depth can improve response time by parallelizing the aggregation work on all non-leaf nodes. This is inspired by the tree-based reduction designs used in MPI [3, 8]. We further extend the previous work by investigating the dynamic formation of a reduction tree for each service call. The tree formation must be load-aware so that the total amount of work for local computation and aggregation is evenly distributed. This allows the response time to be minimized while maintaining sustained throughput. We present the details in the next section.

Tree Formation Scheme	Response Time	Throughput
Base	$\times$	$\times$
Flat	$\times$	$\checkmark$
Hierarchical	$\checkmark$	$\checkmark$

Figure 8: **A qualitative comparison of the three schemes.**

Figure 8 summarizes the performance differences of the above three tree formation schemes. A check mark "$\checkmark$" means the method performs well for achieving the specified objective while a "$\times$" represents the opposite. The Hierarchical scheme performs best both in terms of response times and throughput, while the Base scheme performs worst.

To illustrate the quantitative performance differences of these three schemes, we consider the following special case with simplified assumptions. Suppose all servers are homogeneous while the local processing cost and the binary reduction cost are s and r respectively at each server[1]. Also assume the number of partitions in every DAC request is n. The optimal throughput for the Flat and Hierarchical schemes is estimated as $\frac{n}{ns+(n-1)r}$. This is achieved when all nodes are kept busy all the time. In comparison, the optimal throughput for the Base scheme is estimated as $\min\left\{\frac{1}{nr}, \frac{1}{s}\right\}$. In this case, the consumer becomes the bottleneck when the total aggregation work outweighs its local service work, i.e., $nr > s$. In terms of minimum service response time, the Base and Flat schemes are in the order of $O(n)$ while the Hierarchical scheme is in the order of $O(\log n)$.

4.2 Load-Adaptive Tree (LAT) Formation

The goal of our hierarchical tree formation scheme is to minimize the response time with sustained throughput. Load balance is important in a concurrent execution environment since the response time is always dependent on the slowest node. Thus, our goal can be broken down into two parts: (1) reducing load imbalance when there is high load variation on different nodes; (2) minimizing the response time when load is well balanced. To achieve the above objectives, we design a hierarchical reduction tree by considering the tree *shape* and the *mapping* of servers (service providers) to tree nodes.

Tree shape: We want to control the tree shape for two reasons: (1) The outgoing degree of a node reflects the aggregation work assigned to the node. Thus, it can be leveraged to balance load and indirectly affect the response time. (2) The tree height indicates the critical path length, which directly affects the response time. Therefore, deep tree should be avoided to improve the response time, especially when load is balanced.

Node mapping: Once the shape of a reduction tree is determined, the next step is to map the actual servers to the tree nodes.

[1] We use a uniform cost metric in this calculation even though real costs contain several factors including CPU processing, network and disk I/O. Our definition of operation costs is the consumption of the most critical resource measured in time, e.g., the CPU processing time for a CPU-intensive operation (divided by the number of processors for multi-processor servers).

We introduce a load-aware node placement method in which busy service providers will be mapped to tree nodes with few children nodes. Compared with a baseline random approach in which nodes are mapped randomly, the load-aware approach has two advantages. First, it can be more effective in balancing workload among service providers, especially in a heterogeneous environment. Second, unresponsive or slow servers can be placed at the leaf nodes and they can be discarded if necessary (as will be discussed in Section 4.4).

The load-adaptive tree algorithm: Based on the above discussion, we use two heuristics to guide our algorithm design. (1) The leaf nodes in a reduction tree do not perform aggregation, and thus servers with heavy workload are placed in leaf nodes. On the other hand, partitions with less workload will be placed in interior nodes. In particular, the reduction tree root will be placed at the server with the lightest workload since it does the most amount of aggregation work. (2) When all servers are similarly loaded, the response time is normally determined by the longest path. So longer path should be assigned to partitions with relatively less load.

Our load-adaptive tree formation algorithm dynamically constructs a reduction tree for each DAC request based on the runtime load information on all nodes. It consists of four steps:

(1) **Runtime information collection:** First, we collect the current load on each server, and estimate the costs of the binary aggregation and service operation on each server. These information are used in subsequent steps.

(2) **Assigning reduction operations to nodes:** There are $n-1$ reduction operations to be distributed among n nodes. The assignment is done in a way that less loaded servers will do more aggregation (larger outgoing degree). This can be accomplished step by step by assigning a reduction operation to the least-loaded server. Figure 10 (a) illustrates this process with an example that assigns 7 reduction operations to 8 servers. Numbers on the reduction boxes represent the algorithm steps. For example, at the first step, one reduction is assigned to server A. In the end, server A is assigned four reduction operations.

(3) **Tree construction:** Then we build a load-adaptive tree in the lexicographical breadth-first order. The server that is assigned the most reduction operations is placed to the root and this placement determines the outgoing degree of the root (i.e., a set of unmapped child nodes for the root). Then the following steps are repeated: the server with the most reduction operations among the remaining unplaced servers is placed to the unmapped tree node with the smallest depth. A tie is broken by first placing a server with the higher estimated load. Figure 10 (b) illustrates a LAT derived from Figure 10 (a).

(4) **Final adjustment:** A final step is employed to ensure that the tree height is controlled by $\log n$. We check all subtrees in a breadth-first order and if a subtree is a linked list, i.e., the outgoing degree of each non-leaf node is one, we replace this subtree with a load-aware binomial tree. This tree is built as follows: (1) Sort tree nodes in the descending order of their outgoing degrees, and break ties by using the increasing order of node depths (the root has depth 0). (2) Sort servers based on their workload in ascending order. (3) Match servers with the tree nodes one by one following the above sorted order.

The complete algorithm is summarized in Figure 9. Its time complexity is $O(n \log n)$, which can be achieved using a heap.

The above load-adaptive tree construction ensures the following two properties[2]: (1) If node A is less loaded than node B, then node A will be assigned more children than node B. (2) If node A and node B are of the same depth from the root, and node A is less loaded than node B, then the depth of the subtree rooted from A is larger than the depth of the subtree rooted from node B. That means the algorithm tries to assign less loaded nodes to a longer path.

[2] In the following discussion, we use the term *node* to denote both a certain node in the reduction tree and the server assigned to that node.

1. Collect the current load on each server. Then estimate the costs of a binary reduction operation and service operation on each server.
2. For n nodes, there are $n-1$ reduction operations. We assign them one by one to n servers by repeating the follow steps $n-1$ times.
   - Find the least-loaded server with one additional reduction assigned.
   - Assign one reduction operation to this server and adjust the load estimation on this server.
3. Map servers to tree nodes and form a tree shape based on the above assignment:
   (a) First, we map the server with the most reduction operations as the root and setup its unmapped child nodes.
   (b) Repeat the following steps n-1 times until all servers are mapped.
      - Pick up an unmapped server with the largest aggregation work. If there is a tie, pick up one with the largest total workload.
      - Map this server to an unmapped tree node with the smallest tree depth.
4. Scan all nodes in the tree to ensure the tree height is controlled by $\log n$: Examine each subtree and if it is a linked list, replace this subtree with a load-aware binomial tree.

Figure 9: **LAT: Load-adaptive tree formation algorithm.**

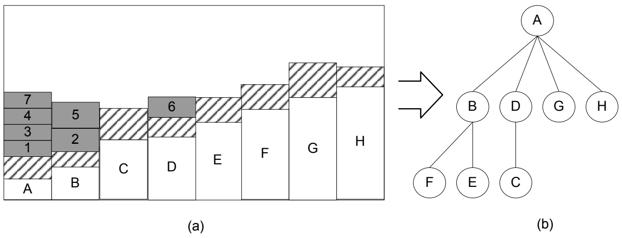

Figure 10: **Constructing an 8-node load-adaptive tree: (a) Assign 7 reduction operations to 8 servers (white boxes - the current workload; stripe boxes - service operation costs; gray boxes - reduction operation costs; numbers show the order of assignments). (b) The result load-adaptive tree.**

4.3 Node-level Concurrency Management

Our initial design allocates one thread to execute the local processing operation and then block waiting for results from other partitions for reduction. When all threads in the fixed-size thread pool are blocked, no further requests can be served even if the system is not busy at all. This situation could be alleviated by increasing the thread pool size. However, a large number of concurrent threads could also degrade system performance substantially [25].

Motivated by previous studies on event-driven concurrency management for Web and network services [14, 25], our solution is to introduce an event driven mechanism for data aggregation support. When a thread needs to wait for results from child nodes, the system registers its interested events and releases this thread. Figure 11 shows the state machine of our event-driven design. The states are

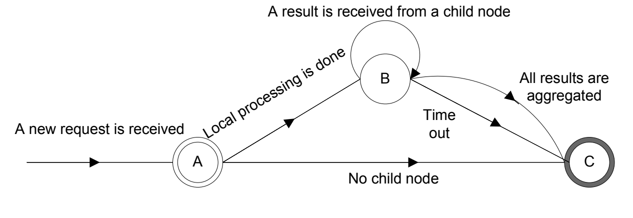

Figure 11: **State machine of event-driven aggregation: (A) Local process initiated. (B) Request partially processed. (C) Results ready to be sent to the parent.**

defined as follows: (1) **State A**: A new request is received and the local processing is initiated. (2) **State B**: The request is partially processed and is pending for results from its children to be aggregated. (3) **State C**: The request is completed locally (i.e., it has aggregated results from all its children), or is timed out; and is ready to be sent to the parent.

Figure 12 shows the node-level architecture for aggregation. It also shows the stages corresponding to the states in Figure 11. When a request comes in, it is placed in a request queue. When it gets its turn to be served, the service function is located in the service library and a thread or a thread-attached process[3] is launched to serve the request. The thread also establishes connections to child nodes based on its position in the reduction tree.

When the service operation is done, the local result from the service operation along with the interested socket events of established connections are placed into an aggregation pool. There is a dedicated thread in the aggregation pool watching on all the registered events. When an event occurs, it will be placed into an event queue along with the partial result. There is also a thread pool associated with the event queue. Events are served in the similar manner as requests. If not all results are received from the child nodes, the partial result is again placed back into the aggregation pool for further processing. Otherwise, the aggregated result will be delivered to the parent node.

Notice that we assign higher priorities to threads serving the event queue than those serving the request queue. This technique improves the mean response time following the shortest job first heuristic.

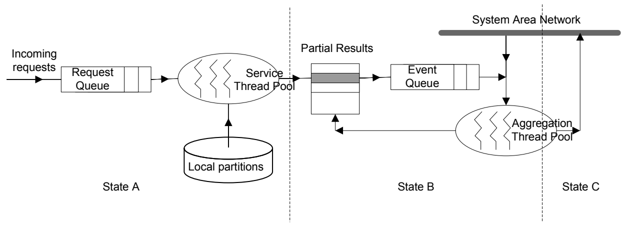

Figure 12: **Node-level aggregation architecture.**

The above procedure is completely transparent to service programmers. A service programmer only needs to provide a service function and aggregation function as stated in Section 3. This achieves the efficiency of the event-driven model while maintaining the simplicity of the thread programming model.

[3] A thread-attached process is a process paired with a thread. The process executes the request while the thread collects the result from the process. This achieves fault isolation without modification of the service library.

4.4 Handling Unresponsiveness and Failures

This section discusses methods to exclude faulty or unresponsive nodes in data aggregation. If a hardware or software component stops working completely, the loss of continuous heartbeat messages allows each node quickly aware of failures. Subsequent DAC invocations will exclude these faulty nodes gracefully from the reduction tree. When services are replicated, failed partitions can be excluded while replicas can be used to achieve fault tolerance. On the other hand, unresponsive nodes may appear normal with periodic heartbeats. When unresponsive nodes are included in a reduction tree, they will not only prolong the overall response time, they could also become the bottleneck and significantly reduce the system throughput.

Our load-adaptive reduction tree design and event-driven request processing is able to alleviate the problems caused by node unresponsiveness to a certain degree. The load-adaptive tree places unresponsive servers in the leaf nodes since they often have the highest observed workload, and thus limit the scope of their adverse effects. The event-driven concurrency management helps the handling of unresponsive nodes by releasing thread resources from being blocked by unresponsive nodes. Therefore, an aggregation node can continue serve other requests although the previous requests are blocked by unresponsive nodes.

We now present another technique called *Staged Timeout* which imposes different soft deadlines on different aggregation nodes to further minimize the impact of unresponsive nodes. We assign a shorter deadline for a node that is closer to the bottom of the tree. In this way, when a node is timed out, its parent still has ample time to pass the partial aggregation results (excluding the results from the failed child) back to the root. The deadline assignment is done in a recursive fashion and can be calculated from the root to the bottom of the tree. Initially, the root of the reduction tree is assigned the same deadline as specified in the DAC request. The deadline for node x with parent y can be recursively calculated as $D_x = D_y - aR_x - \delta$, where D_y is the deadline of node y, and R_x is the rank of node x among its siblings. We rank all siblings of a common parent based on the sizes of subtrees rooted from these sibling nodes (in ascending order). In the formula, δ is a constant to cover various overheads.

In the staged timeout scheme, when there are k unresponsive nodes, we will lose exactly k partitions in the final aggregation results. On the contrary, in the *uniform timeout* scheme where all nodes have the same deadline as the root, k unresponsive nodes will cause $\min\left\{2^{\lceil \log k \rceil + 1} - 1, n - 1\right\}$ partition losses. When k is small, the amount of partition loss under the uniform timeout scheme is about twice as much as that under the staged timeout approach.

4.5 Other Implementation Issues

(1) **Determining Server Workload:** Several previous studies use the request queue length on each node as the load index [26]. We extend it to consider the aggregation work associated with each request. We calculate the load for each request in the queue as $L_i = s + \pi * a$, where s is the mean service time, a is the mean aggregation time and π is the number of reduction operations associated with this request. The summation of request cost (L_i) represents the load index of this server. In terms of aggregation and local service cost, we approximate it using CPU consumption acquired through the Linux `/proc` file system.

(2) **Dealing with Staled Workload Information:** Workload information is disseminated periodically through multicast. Multicast at a high frequency is prohibitively expensive while low frequency multicast results in stale load information [18]. Using staled load

information could lead to flocking effect, i.e., all service requests tend to be directed to the least loaded server between consecutive workload announcements. In our implementation, we use a controlled random correction method for load prediction. First, each node still collects the load information from the multicast channel. Second, the Neptune consumer module randomly polls load information from a few nodes for every service invocation. For other nodes, we take the multicast load information as the base and apply a random correction to it. The deviation of the randomness increases along with the staleness of the multicast load information.

(3) **Reducing Network Overhead:** In our implementation, a service consumer uses multicast to disseminate the reduction tree information along with the actual request to the service providers. We implement a reliable multicast using the reduction tree as the acknowledgment tree. We also use a TCP connection cache to avoid frequent expensive TCP connection set-ups and tear-downs.

5. EVALUATION

The DAC primitive and the runtime support techniques proposed in this paper have been fully integrated in Neptune. Subsequently, we have implemented or optimized several online services using the DAC primitive. We will describe these applications and our evaluation settings in Section 5.1.

Our system evaluation seeks to answer three questions. (1) How easy is it to use DAC to program services (Section 5.2)? (2) How effective is the proposed architecture to reduce response time with sustained throughput? In particular, we assess the system's ability to handle heterogeneous cluster environments, node failures or unresponsiveness (Section 5.3 to Section 5.5). (3) Is the system scalable (Section 5.6)?

5.1 Evaluation Settings

The majority of the evaluation is done through experiments except for the evaluation of system scalability, in which we use simulations for large-scale settings beyond the actual hardware configuration. We describe the (I) applications, (II) hardware platform, (III) workload settings, (IV) simulation model, and (V) evaluation metrics for our evaluation as follows:

(I) Applications: (1) *Search engine document retriever (RET).* The RET service is a prototype document index searching component for a search engine (as we discussed in Section 3.1). It scans through a number of document index partitions and returns an aggregated list of document identifications that are relevant to a certain query. (2) *BLAST protein sequence matcher (BLAST).* The BLAST service is based on NCBI's BLAST [1] protein sequence matching package. The DAC primitive helps this application to speed up the lengthy matching process over a partitioned protein database. (3) *Online facial recognizer (FACE).* The FACE service reassembles the case where cameras at airport security checkpoints take pictures of passengers, and compare them at real time against an image database of wanted criminals. The similarity between two facial images is calculated using the eigenface algorithm [22]. The image database is partitioned and we use DAC to facilitate fast response, which is very critical to avoid delaying passengers. (4) *Microbenchmark (MICRO).* In addition to the above three applications, we also implemented a microbenchmark application, in which we use CPU spinning with different lengths for local processing and global reduction operations. An advantage of using this microbenchmark is that we can control service granularities and isolate application-specific artifacts.

(II) Hardware platform: All the experimental evaluations were conducted on a rack-mounted Linux cluster with 30 dual 400 Mhz Pentium II nodes (with 512MB memory) and 4 quad 500 Mhz Pentium II nodes (with 1GB memory). Each node runs RedHat Linux (kernel version 2.4.18), and has two Fast Ethernet interfaces. All nodes are connected by a Lucent P550 Ethernet switch with 22 Gb/s backplane bandwidth.

(III) Workload settings: Due to space constraint, our experimental evaluation mainly focuses on two of the four applications – RET and MICRO. Our extended study showed that results obtained from these two applications are quite representative. For RET, the service data are divided into 28 partitions (24 on dual-CPU nodes and 4 on quad-CPU nodes), each of which is between 1GB to 1.2GB (and cannot be completely fit in memory). The evaluation is driven by a trace obtained from `http://www.ask.com/`, which contains query terms and timestamps[4]. The trace contains only uncached queries and exhibits little temporal locality. We proportionally adjust request arrival intervals to generate desired request demand. The RET service is fairly coarse grained and the observed maximum throughput of RET is below 20 req/sec on a dual-CPU node. Therefore, we choose the settings of MICRO to represent a fine-grain service. The spin times for the local processing operation and global reduction operation follow the exponential distribution, with their means being $25ms$ and $5ms$ respectively. We also model the request arrival as a Poisson process for MICRO. The soft deadline guarantees for both services are set to be 2 seconds.

(IV) Simulation model: Our simulator is in fact a by-product of our architecture design and has been extensively used to aid choices of various design alternatives (such as reduction tree schemes or staged timeout policies). The simulation model closely reassembles the real situation of the MICRO service. Request arrival is modeled as a Poisson process. Each service node is modeled as a multi-processor node with two non-preemptive FIFO task queues (one for reduction operations and one for local processing operations). In the following simulations, all servers have been configured with two processors. The service and reduction time follow exponential distributions. The TCP and UDP packet delays in our simulation are $170\mu s$ and $145\mu s$ respectively, following real measurement results.

(V) Evaluation metrics: Two performance metrics are used in our evaluation. (1) *Response time.* This is the average response time of *successful requests* – requests that are completed within the specified deadlines and meet the service quality requirements. (2) *Throughput.* We use both the conventional throughput metric and an enhanced *quality-aware* throughput in our performance study. The quality-aware throughput is introduced to measure the system performance under node failures or unresponsiveness and it will be described in Section 5.5 in more detail.

5.2 Ease of Use

We evaluate the usability of DAC through the amount of programming effort on the data aggregation parts of the four applications we have implemented. For this purpose, we first implemented (or ported) the four applications under Neptune and used a client-side loop to perform data aggregation over a set of partitions. We then let a graduate student, who has moderate familiarity with Neptune but has never used the DAC primitive before, optimize the data aggregation loop using DAC. We report the code size change after the optimization, and the amount of time spent on the optimization (including the debugging time). The results are shown in Figure 13, in which we also list the original code sizes of the applications. As we can see, little effort is required to optimize data aggregation operations using DAG. Specifically, the code size increase ranges from 70 to 300 lines, and it takes at most two days to learn the DAC primitive, revise the code, and debug them. These results demonstrate that DAC is easy to learn and use.

[4] IP addresses are filtered out for privacy reasons.

Service	Code Size	Code Size Change	Programming Effort
RET	2384 lines	142 lines	1.5 days
BLAST	1060K lines	307 lines	2 days
FACE	4306 lines	190 lines	1 day
MICRO	400 lines	77 lines	3 hours

Figure 13: Ease-of-use of the DAC primitive.

5.3 Tree Formation Schemes

In this section, we compare the impact of different tree formation schemes on system performance. Particularly, we demonstrate that the load-adaptive tree formation scheme performs the best to reduce response time with sustained throughput, for both homogeneous and heterogeneous environments.

We compare among four tree formation schemes. (1) **Base**: the baseline scheme where data aggregation is performed by the service consumer. (2) **Flat**: all participating nodes form a flat tree whose root is randomly picked. (3) **Binomial**: all participating nodes form a binomial tree and they are assigned to tree nodes randomly. (4) **LAT**: our load-adaptive tree formation scheme.

We first show the results under a homogeneous setting with 24 dual-CPU nodes. Figure 14 and Figure 15 show the system throughput and response time respectively as functions of the incoming request rate. Each figure contains two graphs corresponding to the MICRO and RET services respectively. As we can see, for both services, **Base** performs the worst because all requests flow through the same root which overwhelms the root node. As a result, the throughput of **Base** quickly drops to close to zero and the response time increases to the deadline. For the remaining three schemes, they perform similarly when the request demand is low; and when the request demand is high, **Flat** performs the worst and **LAT** performs the best for both throughput and response time. Additionally, the advantage of **LAT** is more evident in terms of response time over the other two schemes. Specifically, for MICRO, the response time of **LAT** is up to 38.7% better than **Binomial** and 39.8% better than **Flat**; for RET, the response time of **LAT** is up to 16.3% better than **Binomial** and 39.0% better than **Flat**. These results confirm that our load-adaptive tree shape design is effective to reduce response time with sustained throughput.

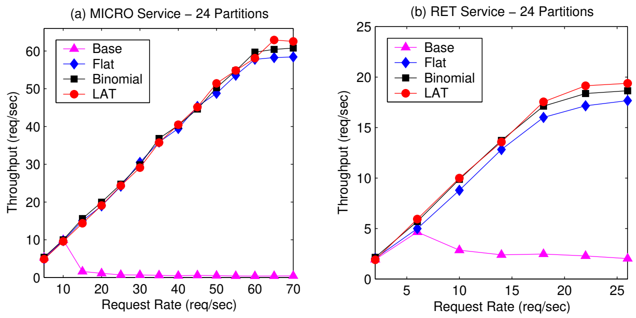

Figure 14: System throughput under different tree formation schemes in a homogeneous environment.

We further compare the tree formation schemes in a heterogeneous setting with 20 dual-CPU nodes and 4 quad-CPU nodes. The goal of this experiment is to show that **LAT** is even more effective to balance load and reduce response time in heterogeneous environments. We did not show the results for **Base**, which performs too poor to make it relevant to this study. The results are shown in Figure 16 (throughput) and Figure 17 (response time).

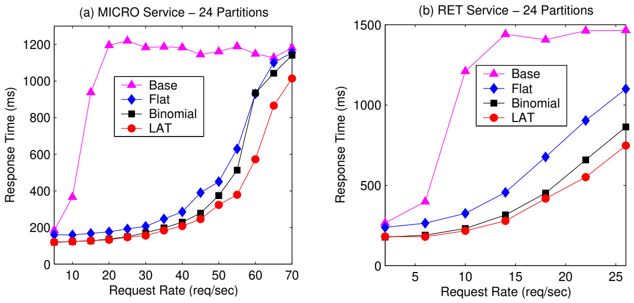

Figure 15: Response time under different tree formation schemes in a homogeneous environment.

As we can see, **LAT** again outperforms **Binomial** and **Flat** for both throughput and response time. Particularly, the throughput differences between **LAT** and the other two schemes become more evident than the results in a homogeneous setting (Figure 14). This is due to the fact that **LAT** is able to make use of the extra processing power in the quad-CPU nodes by assigning more workload to those nodes. Specifically, for MICRO, the throughput of **LAT** is up to 22.5% better than **Binomial** and 29.3% better than **Flat**; for RET, the throughput of **LAT** is up to 21.0% better than **Binomial** and 29.9% better than **Flat**. Additionally, the response time differences between **LAT** and the other two schemes are also enlarged. This is because **LAT** is more effective to balance load on all nodes, and thus reduces the latency of the critical path in a reduction tree, which is determined by the most loaded node. Specifically, for MICRO, the response time of **LAT** is up to 55.1% better than **Binomial** and 62.3% better than **Flat**; for RET, the response time of **LAT** is up to 25.3% better than **Binomial** and 54.8% better than **Flat**. These results confirm that our load-adaptive tree shape design is even more effective in a heterogeneous setting.

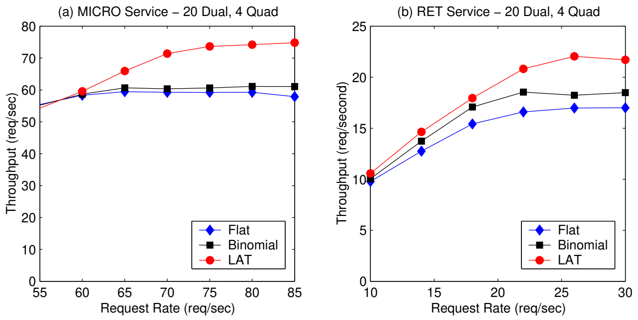

Figure 16: System throughput under different tree formation schemes in a heterogeneous environment.

In summary, our **LAT** tree formation scheme is effective to minimize response time with sustained throughput. And its advantage becomes more evident in a heterogeneous environment.

5.4 Event-driven Aggregation

In this section, we evaluate the effectiveness of the event-driven aggregation mechanism. We compare our system (**ED**) with a modified scheme in which worker threads are blocked while waiting for results from their children. We call the second scheme **NoED**. We run the MICRO service on 24 dual-CPU nodes. The results are shown in Figure 18 (throughput) and Figure 19 (response time). As we can see from Figure 18, when the incoming request rate is

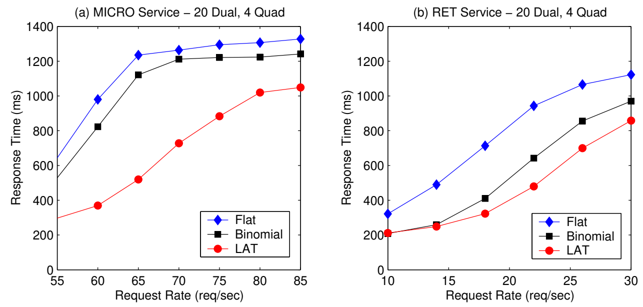

Figure 17: Response time under different tree formation schemes in a heterogeneous environment.

low, both **ED** and **NoED** performs similarly. However, when the request rate grows beyond a certain point, the throughput for **NoED** plunges. This is because it takes longer to serve each DAC request when the request demand increases, which will cause worker threads on each node to be blocked for a longer time. When the request demand grows beyond a certain point, some nodes might even become idle because all threads are blocked waiting for responses from their children. This further reduces the system's capacity to process DAC requests, and could eventually lead to deadlock. Deadlocks are not released until the 2-second deadline is reached. Additionally, from Figure 19, we can see that the response time under **NoED** also increases dramatically after the plunging point. These results demonstrate that the event-driven aggregation design improves the system concurrency and is critical to maintain system throughput under heavy load.

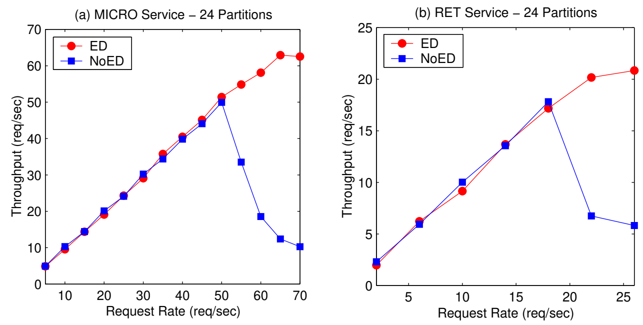

Figure 18: System throughput with/without event-driven aggregation.

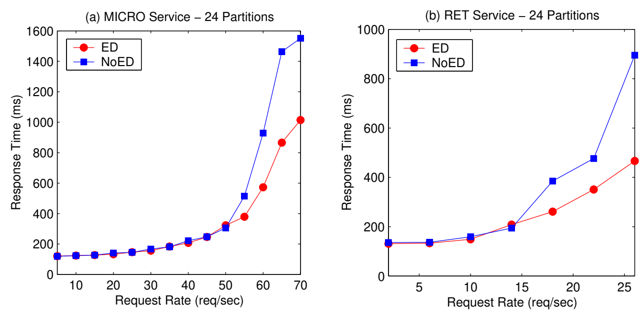

Figure 19: Response time with/without event-driven aggregation.

5.5 Handling Node Unresponsiveness

The goal of this experiment is to show that our proposed architecture design is effective to handle node unresponsiveness. The techniques employed in the architecture design can be broken down into two categories according to their roles in the handling of node unresponsiveness. The first category consists of the load-adaptive tree formation scheme and the staged timeout policy. The load-adaptive tree formation scheme assigns unresponsive servers to leaf nodes. In addition to that, the staged timeout policy causes these unresponsive leaf nodes to be timed out earlier and thus excludes them from the reduction tree. Overall, the combination of these two techniques eagerly prunes the unresponsive nodes from the reduction tree, thus we call it *Eager-Pruning* or *EP*. Without EP, an unresponsive node could cause multiple node timeouts as discussed in Section 4.4. The second category consists of the event-driven request processing scheme. Event-driven design prevents threads on healthy nodes from being blocked by its slow or unresponsive children. We call the second category *Event-Driven* or *ED*. Without ED, an unresponsive node would block a worker thread on its ancestor nodes until the timeout period expires.

We evaluate the effectiveness of these techniques by comparing four schemes: (1) No Eager-Pruning and no Event-Driven (**None**). (2) Eager-Pruning without Event-Driven (**EP only**). (3) Event-Driven without Eager-Pruning (**ED only**). (4) Eager-Pruning and Event-Driven (**EP+ED**). Note that **EP+ED** corresponds to the real implementation. In schemes with no Eager-Pruning, we use the binomial tree scheme with random node assignment, and all nodes in a reduction tree have the same timeout value.

Different schemes may exhibit different capability to retain aggregation qualities in the event of node unresponsiveness (i.e., to minimize the partition losses from the final results). To reflect this fact, we use the metric of *quality-aware throughput* instead of the plain throughput metric. The quality-aware throughput is defined as a weighed throughput where the weight is the aggregation quality for each request. For example, a request that returns the aggregation result of 90% of the total partitions will be counted as 0.9 toward the quality-aware throughput.

We run the MICRO service with 24 dual-CPU nodes, and measure the quality-aware throughput over a period of 60 seconds. Each data point is measured as the quality-aware throughput over the past two seconds. During the whole period, two nodes become unresponsive and then recover. The first node becomes unresponsive at second 10 and recovers at second 30; and the second node becomes unresponsive at second 20 and recovers at second 40. The effect of node unresponsiveness is emulated by increasing the processing time or reduction time of each request by 5 folds. The incoming request rate is at 45 req/sec, which corresponds to 75% system capacity level.

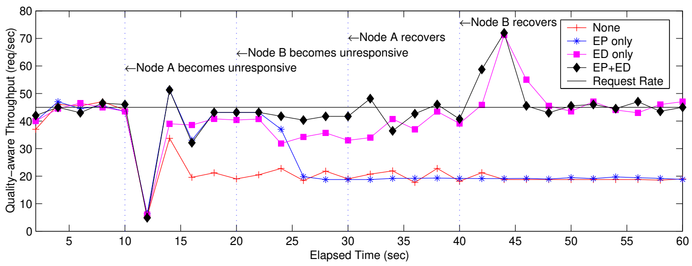

Figure 20: Effectiveness of staged timeout scheme to handle unresponsive nodes.

The experimental results are shown in Figure 20. As we can see, all schemes perform similarly when there is no node failure (before second 10). Right after the first node failure, the quality aware throughput of all schemes plunges (second 12). This is caused by the transient effect that all nodes are waiting for results from the unresponsive node until the 2-second timeout period expires. After second 12, **None** performs the worst among the four schemes because it suffers a high percentage of partition losses in each request and exhibits low concurrency without using either EP or ED. **EP only** improves from **None** during the period when only one node is unresponsive (second 14 to 24), this is because EP places the only unresponsive node as the direct child of the root node, and thus limits the adverse effect of the unresponsive node (i.e., it blocks only one worker thread on the root node). When the second node fails, the unresponsive nodes quickly cause more working threads to be blocked and lead to very low concurrency. On the other hand, both **ED only** and **EP+ED** maintains high concurrency through event-driven request processing, and thus can achieve much better throughput even when two nodes become unresponsive. **EP+ED** outperforms **ED only** by limiting the losses to only the unresponsive nodes and thus improving the quality of processed requests.

The four schemes also exhibit different behaviors when both nodes recover. **ED only** and **EP+ED** has a throughput surge due to the residual effect that all old requests pending for timeout now suddenly complete. On the other hand, the quality-aware throughput of **None** and **EP only** remain at a low level. This is caused by the fact that during the node unresponsiveness, the task queues in both healthy nodes and unresponsive nodes grow excessively long; and it takes a long time to clean up these already expired requests.

In conclusion, our architecture design is effective to handle node unresponsiveness by eagerly pruning unresponsive nodes and using event-driven request processing to avoid unresponsive nodes from blocking worker threads.

5.6 Scalability Study

In this section, we verify the scalability of our architecture design. We first compare the experimental results with simulation results under small-scale settings and show that the simulation results closely conform to the experimental results. Then we use simulation to access the scalability of our architecture design under large-scale settings.

Figure 21 shows the simulation results and experimental results for the MICRO service for small-scale settings with 4 to 24 partitions. We measure the system throughput and response time with the request rate at 30 req/sec (corresponding to 50% system capacity level). As we can see, the predicted throughput closely matches the experimental results; and the predicted response time differs from the experimental results by only a small constant. The latter is mainly due to the fact that the simulator does not cover all system overhead. This provides us with high confidence for relying on simulations to predict the scalability of our architecture design.

Figure 22 shows the large-scale simulation results. We vary the number of nodes from 4 to 512. We measure the system throughput, and the response times under 50%, 60%, 70%, 80% and 90% demand levels. As we can see, the maximum throughput varies little when the number of nodes increases, and is very close to the ideal throughput[5] Additionally, the response time grows logarithmically with the increase of the number of nodes. These results show that our architecture is scalable to a large number of nodes.

[5] The ideal throughput P can be calculated by $\frac{nk}{ns+(n-1)r}$, where n is the number of partitions, k the number of processors per node, and s and r the service times for the local processing operation and the global reduction operation respectively.

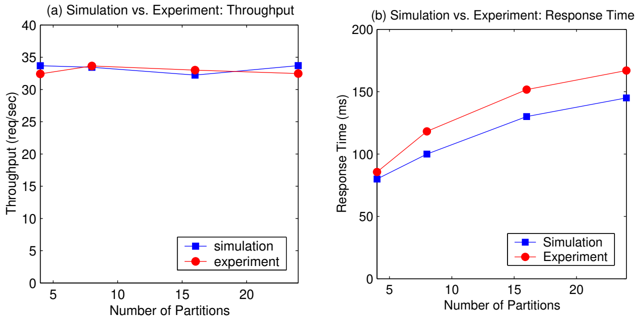

Figure 21: Simulation results in small scale.

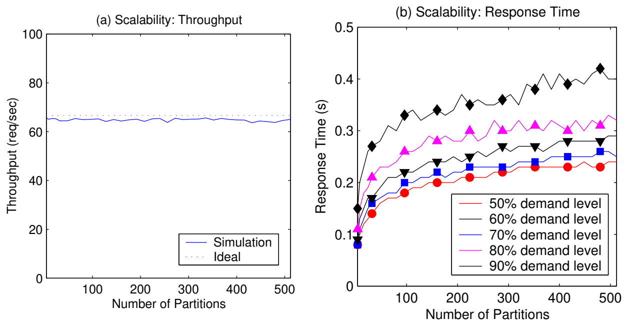

Figure 22: Simulation results in large scale.

5.7 Evaluation Summary

In summary, our evaluation finds out: (1) The DAC primitive is easy to use for implementing data aggregation operations. (2) The **LAT** tree formation scheme is able to deliver low response time and high throughput in both homogeneous and heterogeneous environments, in comparison with other tree formation schemes. (3) The event-driven request processing is effective in maintaining acceptable throughput under system overload. (4) The combination of load-aware tree formation, event-driven request processing, and staged timeout policy is able to gracefully handle node unresponsiveness. (5) Our architecture design is scalable to maintain system throughput with logarithmic increase of response time for data aggregation over a large number of partitions.

6. RELATED WORK

This work is a continuation of our previous research on Neptune: a cluster-based software infrastructure for aggregating and replicating partition-able network services [17, 19]. It is closely related to a group of studies on building cluster-based network services, such as TACC [6], MultiSpace [7], and Ninja [24]. For instance, TACC supports the transformation, aggregation, caching, and customization for the construction of scalable Internet services [6]. The design of Neptune coincides with these systems in several aspects including the single program multiple connection model and implicit concurrency principle. Our work described in this paper complements these systems with efficient programming and runtime support for data aggregation operations. Although such support is currently built as part of the Neptune middleware system, the techniques are equally applicable to other software infrastructures for supporting cluster-based Internet services.

A number of message or stream-based service programming paradigms are supported in the Tuxedo system [23]. However, Tuxedo does not provide direct programming or runtime support for data aggregate operations. Event-driven request processing has been studied in Flash [14] and SEDA [25]. Flash specifically targets the construction of efficient Web servers and SEDA requires application developers to explicitly program in an event-driven model. In either case, significant additional effort may be needed for supporting new applications. Our design takes advantage of the semantics of the DAC primitive and encapsulates the state machine transition design inside the infrastructure. Thus our DAC primitive exposes an easy-to-use interface and at the same time it can achieve the efficiency offered by the event-driven concurrency management.

MPI [20] also supports data reduction operations and several previous works have studied tree-based MPI reductions [8, 9, 10, 21]. Our DAC primitive targets service programming with different concerns. MPI reduction does not concern about throughput optimization, node failure, and deadline guarantees. Also previous tree-based MPI reduction studies mainly focus on static tree shapes. In contrast, our load-adaptive tree formation scheme dynamically constructs the reduction tree using runtime load information.

Internet search services such as Google, Inktomi, and Teoma/Ask Jeeves have implemented their customized data aggregation. While there is no publication on these efforts, our goal is to provide a more general framework for scalable data aggregation. Data aggregation has also been studied in distributed database research [16] and recently for wireless ad-hoc sensor networks [11]. These studies focus on SQL-based data aggregations while our DAC primitive targets more general aggregation operations. Previous studies have proposed and evaluated various load balancing policies for cluster-based distributed systems [12, 18, 26]. These studies target load balancing for service accesses each of which can be fulfilled at a single node or a single data partition. These results can not be directly used for supporting data aggregation operations that involve significant inter-node communication and synchronizations.

7. CONCLUDING REMARKS

This paper presents the design and implementation of the Data Aggregation Call (DAC) primitive to exploit partition-based parallelism in Internet services to support scalable data aggregation operations. Our architecture design leverages load information and hierarchical tree shapes to improve response time with sustained throughput in both homogeneous and heterogeneous environments. Furthermore, several techniques are developed to handle unresponsive nodes. We have successfully implemented several real applications with the DAC primitive. Our experimental and simulation results demonstrate the ease-of-use of the DAC primitive, the effectiveness of proposed techniques, and the scalability of our architecture design.

Acknowledgment. This work was supported in part by NSF ACIR-0082666, 0086061, and EIA-0080134. We would like to thank the anonymous referees for their valuable comments and help.

8. REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proc. of the 10th IEEE Intl. Parallel Processing Symposium*, Honolulu, HI, Apr. 1996.

[3] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient Collective Communication on Heterogeneous Networks of Workstations. In *Proc. of International Conference on Parallel Processing*, 1998.

[4] C. Chang, T. Kurc, A. Sussman, U. Catalyurek, and J. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *SIAM PPSC*, Portsmouth, Virginia, Mar. 2001.

[5] A. Fox and E. A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proc. of HotOS-VII*, Rio Rico, AZ, Mar. 1999.

[6] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *ACM SOSP*, Saint Malo, Oct. 1997.

[7] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. In *USENIX Annual Technical Conf.*, Monterey, CA, June 1999.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[9] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'2000)*, Cancun, Mexico, May 2000.

[10] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *ACM PPoPP*. ACM, May 1999.

[11] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, Boston, MA, Dec. 2002.

[12] M. Mitzenmacher. On the Analysis of Randomized Load Balancing Schemes. In *Proc. of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 292–301, Newport, RI, June 1997.

[13] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *ACM ASPLOS*, San Jose, CA, Oct. 1998.

[14] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proc. of 1999 Annual USENIX Technical Conf.*, Monterey, CA, June 1999.

[15] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability, and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *ACM SOSP*, Charleston, SC, Dec. 1999.

[16] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*, San Jose, CA, USA, May 1995.

[17] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[18] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-grain Network Services. In *Proc. of International Parallel & Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002.

[19] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, Mar. 2001.

[20] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[21] H. Tang and T. Yang. Optimizing Threaded MPI Execution on SMP Clusters. In *Proc. of 15th ACM International Conference on Supercomputing*, Naples, Italy, June 2001.

[22] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Neuroscience*, 3(1):71–86, 1991.

[23] WebLogic and Tuxedo Transaction Application Server White Papers. http://www.bea.com/products/tuxedo/papers.html.

[24] J. R. von Behren, E. A. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A Framework for Network Services. In *Proc. of 2002 Annual USENIX Technical Conf.*, Monterey, CA, June 2002.

[25] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *ACM SOSP*, Banff, Canada, Oct. 2001.

[26] S. Zhou. An Experimental Assessment of Resource Queue Lengths as Load Indices. In *Proc. of the Winter USENIX Technical Conf.*, pages 73–82, Washington, DC, Jan. 1987.