Competitive algorithms for the dynamic selection of component implementations

by D. M. Yellin

As component-based development matures, more and more applications are built by integrating multiple distributed components. We suggest providing components with multiple implementations, each optimized for a particular workload, and augmenting the component run-time environment with a mechanism for switching between implementations. This mechanism monitors the types of requests the component is receiving, and adaptively switches implementations for optimal application performance. Achieving this optimal performance depends on making good choices as to when and how to switch implementations, a problem we refer to as the *adaptive component problem*. We first formalize the generic problem and then provide an algorithm, named Delta, for switching implementations in the special case when the component has exactly two implementations. We show that this algorithm is (3 + epsilon)-competitive with respect to the optimal algorithm, where epsilon is a small fraction. We establish a 3-competitive lower bound for the problem, which implies that Delta is close to optimal. We describe the application of these results to the distributed pub/sub problem, and the data structure selection problem.

Many applications are being built by integrating multiple distributed components in order to implement a particular business function. The increasing popularity of component programming models, such as JavaBeans**[1] and Web services,[2] is predicted to further accelerate the adoption of *component-based development* (CBD).

An impediment to the adoption of CBD, however, is the inability of the "user" of the components to optimize their performance for use in a particular solution. Web services, with its premise of loosely coupled distributed components working together, makes this issue even more acute, as the development, deployment, and maintenance of components making up a single solution may come from different vendors and run in very different system environments.

In this paper, we propose a generic framework that addresses this issue. In our formulation, an *adaptive component* has multiple implementations, each optimized for a particular request workload. A mechanism for switching between implementations is also provided. The underlying system monitors the current request workload for a given component and adaptively switches to the implementation best suited to this workload. By shifting more of the performance optimization burden to the component, performance tuning is simplified and the resulting overall system performance is likely to be improved. Additionally, by having the system monitor the request workload and dynamically switch implementations based upon

current usage patterns, the component can better accommodate changing request patterns.

Because switching between implementations can incur a heavy cost, good algorithms are needed for determining, at run time, when to switch between implementations. We call this the *adaptive component problem*. This paper describes an algorithm, named Delta, for the case when there are exactly two implementations. We show that Delta is $(3 + \epsilon)$-competitive. This algorithm is designed for the case when the cost of switching between implementations is very large compared to the cost of processing a single request. In this case, the value of ϵ is guaranteed to be very small. Because we also show a 3-competitive lower bound, this algorithm is close to optimal.

We show the applicability of this framework to two problems. The first is an adaptive version of the distributed *pub/sub* problem, where multiple loosely coupled components are reading and writing from a shared data repository. A component can either read and write to this shared repository or create a local data cache for fast access. The second example is an adaptive version of the *data structure selection* problem, where an application must choose the appropriate internal data structure to use in order to provide the quickest answer to particular queries. We show that both of these are instances of the adaptive component problem and that Delta can be used to decide when to switch implementations, thus optimizing run-time performance. This illustrates the applicability of our framework to a wide range of problems.

Here is an outline for the rest of this paper. The next section "The adaptive component problem" is followed by "The Delta algorithm." Then, the section "Examples" contains our two case studies. In the section "Competitiveness" we prove that Delta is $(3 + \epsilon)$-competitive, and in the section "Lower bounds," we establish a 3-competitive lower bound for this problem. The section "Related work" is followed by "Summary and open issues."

The adaptive component problem

We propose a model where the component developer implements multiple versions of a component, each optimized for a specific request workload. The developer also writes the code that controls the switching between implementations at run time. For each request type that the component can service, and for each implementation of that component, we determine the cost for processing a request of that type. We similarly determine the cost of switching between any two implementations. These costs may be specified by the component developer or may be derived empirically (e.g., by profiling). An on-line algorithm will monitor request workloads at run time and determine when to switch implementations. The rest of this section formalizes these concepts and defines an optimality criterion for determining when to switch implementations.

Let *Comp* be an adaptive component, and let $ReqTypes = \{typ_i\}$ be the set of request types that *Comp* can process. Let $Impls = \{impl_j\}$ be the set of implementations of *Comp*, with $impl_1$ being the *default* implementation. Let $Cost : ReqTypes \times Impls \rightarrow \mathbf{R}$ be the function that gives the cost for *Comp* to process a request of a given type using a particular implementation ($\mathbf{R}$ denotes the set of reals). Let $SwitchCost : Impls \times Impls \rightarrow \mathbf{R}$ be the function that determines the cost in *Comp* of switching from one implementation to another. $SwitchCost(impl_i, impl_j) = \infty$ iff *Comp* cannot switch from $impl_i$ to $impl_j$. The cost functions may reflect internal computation costs, network message costs, or a combination of these and other metrics, depending upon the application environment.

We represent the computation to be performed as a sequence of requests, $r_1, \cdots, r_k$. The empty sequence is denoted by Ω. To facilitate the modeling of many different sorts of problems, we allow a single request to be processed by either a single component or by multiple components. Given such a sequence of requests, a switch of implementations may occur after processing any request in the sequence. We represent this by the operation $switch(impl_i, impl_j)$, where $impl_i$ is the current implementation of the component, and $impl_j$ is the implementation being switched to. Hence, given a sequence of requests $\alpha = r_1, \cdots, r_k$, we model the adaptive behavior of the component *Comp* as transforming this sequence to the sequence $\alpha' = s_1, \cdots, s_f$ such that the following comments are true:

- For $1 \leq i \leq f$, either s_i is a request or s_i is a switch operation.
- Removing the switch operations from α' produces α.
- If $s_m = switch(impl_j, impl_k)$ then either s_m is the first switch operation in the sequence and $j = 1$ or the closest preceding switch operation in the sequence is of the form $switch(impl_i, impl_j)$.

An *on-line* algorithm is one that transforms the sequence α by deciding whether or not to insert a switch operation after request r_i based only upon the sequence seen so far, $r_1, \cdots, r_i$. We write $\alpha \rightarrow_A \alpha'$ to indicate that on-line algorithm A (used by *Comp*) maps α into α' when processing it. For any request s_i in the transformed sequence α', we say that the *implementation* $impl_k$ *is active at* s_i if the closest *switch* operation preceding s_i in the sequence is of the form $switch(impl_j, impl_k)$, or $k = 1$ and there is no *switch* operation preceding s_i. When algorithm A and sequence α are understood from the context, we simply denote this by $ImplIs(s_i, impl_k)$.

$Cost^A_\alpha$ is the cost of *Comp* to process request sequence α using algorithm A. $Cost^A_\alpha = Cost(\alpha') = \sum_{i=1}^{n} C(s_i)$ where $\alpha \rightarrow_A \alpha' = s_1, \cdots, s_n$ and where

$$C(s_i) = \begin{cases} Cost(s_i, impl_k) \\ \quad \text{if } s_i \text{ is a request, and } implIs(s_i, impl_k); \\ SwitchCost(impl_j, impl_k) \\ \quad \text{if } s_i \text{ is the operation } switch(impl_j, impl_k). \end{cases}$$

Let O be an optimal algorithm; that is, for any sequence α and any on-line algorithm A, $Cost^O_\alpha \leq Cost^A_\alpha$. An on-line algorithm A is *c-competitive*[3,4] iff, for any sequence α, there exist constants c and d such that $Cost^A_\alpha \leq c * Cost^O_\alpha + d$. Given *Comp, RegTypes, Impls*, *Cost*, and *SwitchCost*, the *adaptive component problem* is to find an on-line competitive algorithm for this problem instance, that is, to find a *c-competitive* algorithm for some constant c.

Now consider a special case of the adaptive component problem where $Impls = \{impl_1, impl_2\}$, which we call the adaptive *two-implementation-component problem* (the adjective "adaptive" is sometimes omitted, for conciseness). We assume that there is at least one request r such that $Cost(r, impl_1) < Cost(r, impl_2)$, and at least one request r' such that $Cost(r', impl_1) > Cost(r', impl_2)$. Also, we assume that there are no constraints on the order in which requests must be composed in a sequence (i.e., any interleaving of requests is a legitimate request sequence).

Let $SC_1 = SwitchCost(impl_1, impl_2)$, $SC_2 = SwitchCost(impl_2, impl_1)$, and $SC = SC_1 + SC_2$. We call SC the *round trip switching cost*. The Delta algorithm described in the next section will perform close to optimal when, for any request r, $|Cost(r, impl_1) - Cost(r, impl_2)| \ll SC$. That is, the difference in processing cost for a single request when one implementation is active as opposed to the other implementation, is significantly less than the switching cost.

The Delta algorithm

Consider any two-implementation-component problem, where SC is the previously defined round trip switching cost. Let $\alpha = r_1, \cdots, r_k$ be a sequence of requests. Then $Cost(\alpha, impl_j) = \sum_{i=1}^{k} Cost(r_i, impl_j)$, $j = 1, 2$. Algorithm Delta is extremely simple and works as follows. Say that implementation $impl_i$ of component *Comp* is active, $i \in \{1, 2\}$, and the processing of request r_k in the sequence $r_1, r_2, \cdots$ has just ended. Given $impl_i$, denote the other implementation by $\overline{impl_i}$. If there exists $j \leq k$ such that $Cost((r_j, \cdots, r_k), \overline{impl_i}) \leq Cost((r_j, \cdots, r_k), impl_i) - SC$, then *Delta* instructs *Comp* to switch implementations.

Delta also has a simple implementation using just three counters: *Impl*1*Cost*, *Impl*2*Cost*, and *MinDelta*. The algorithm, when $impl_1$ is assumed active, is given in Figure 1. The algorithm works analogously when $impl_2$ is active.

To see why this implementation is correct, note that by definition $Cost(r_0) = 0$ and if Delta switches to $impl_2$ after r_k, then $\exists j$, $1 \leq j \leq k$, such that

$$\begin{aligned} &(Cost((r_1, \cdots, r_k), impl_1) \\ &\qquad - Cost((r_0, \cdots, r_{j-1}), impl_1)) \\ &-(Cost((r_1, \cdots, r_k), impl_2) \\ &\qquad - Cost((r_0, \cdots, r_{j-1}), impl_2)) \geq SC \\ &\Leftrightarrow \\ &(Cost((r_1, \cdots, r_k), impl_1) \\ &\qquad - Cost((r_1, \cdots, r_k), impl_2)) \\ &-(Cost((r_0, \cdots, r_{j-1}), impl_1) \\ &\qquad - Cost((r_0, \cdots, r_{j-1}), impl_2)) \geq SC \end{aligned}$$

After processing request k in the sequence, the value of *MinDelta* in the implementation of Figure 1 equals the minimum of 0 and $\min_{0 \leq j \leq k} \{Cost((r_0, \cdots, r_j), impl_1) - Cost((r_0, \cdots, r_j), impl_2)\}$.

At this same point *Impl*1*Cost* equals $Cost((r_1, \cdots, r_k), impl_1)$, and *Impl*2*Cost* equals $Cost((r_1, \cdots, r_k), impl_2)$. Hence the equation above is true iff:

$$(Impl1Cost - Impl2Cost) - MinDelta \geq SC$$

This is exactly the computation that the algorithm of Figure 1 performs in order to determine when to switch implementations.

Figure 1 The Delta Algorithm ($impl_1$ assumed active)

```
Impl1Cost := 0;  Impl2Cost := 0; MinDelta := 0;
TimeToSwitch := false;
While (not TimeToSwitch) {
        Process next request r;
        Impl1Cost := Impl1Cost + Cost(r,impl1);
        Impl2Cost := Impl2Cost + Cost(r,impl2);
        Temp := Impl1Cost – Impl2Cost;
        MinDelta := min(MinDelta,Temp);
        If ((Impl1Cost – Impl2Cost – MinDelta) > = SC)
          {TimeToSwitch := true;}
}
Switch to impl2;
```

The section "Competitiveness," later, contains a formal proof that Delta is competitive and close to optimal. We now provide some intuition as to why this is the case. Let A be any algorithm and consider B, the "adversary" of A, an algorithm that observes A and tries to devise a sequence on which A performs worse than B. If A keeps $impl_1$ active too long, the adversary will devise the sequence to be costly when $impl_1$ is active and cheap when $impl_2$ is active. Hence, any competitive algorithm must keep switching between implementations when it determines that the cost of keeping the other implementation active is lower. But if A switches implementations too often, the switching costs will dominate and the adversary can choose to keep the same implementation active, thus avoiding all switching costs. The key insight of Delta is to switch exactly when it accumulates SC more in cost than the other implementation would, if it were active. Choosing any value other than SC results in worse performance.

To see why this is the case, say that A makes $impl_1$ active and switches to $impl_2$ after accumulating additional cost k, $k < SC$ (more than it would have accumulated if $impl_2$ were active). As soon as A makes $impl_2$ active, the adversary would design the sequence so that A accumulates additional cost k when $impl_2$ is active. At that point A will switch back to $impl_1$. The adversary would keep $impl_1$ active throughout this sequence. Hence, the cost to the adversary would be just k, while A's cost would be $k + SC + k > 3k$. A would thus be worse than 3-*competitive* (how much worse would depend upon the choice of k).

On the other hand, say that A makes $impl_1$ active and A switches to $impl_2$ after accumulating additional cost k, $k > SC$. The adversary would make $impl_2$ active at the start of the sequence. As soon as A accumulates k in cost and switches to $impl_2$, the adversary would switch to $impl_1$ and would design the sequence so that A accumulates additional cost k when $impl_2$ is active. At that point A will switch back to $impl_1$. Hence, the cost to the adversary would be just SC, while A's cost would be $k + SC + k > 3 * SC$. Hence, A would be worse than 3-*competitive*.

The reason Delta is $(3 + \epsilon)$-competitive and not just 3-competitive is due to *boundary* conditions; Delta does not always have the opportunity to switch when it accumulates exactly SC more in cost. For instance, in may be that $impl_1$ is active and Delta has accumulated $SC - 1$ more in cost than it would have if $impl_2$ had been active. The next request r may cost it $rCost = max_r Cost(r, impl_1) - Cost(r, impl_2)$ more to process in $impl_1$ than in $impl_2$. Therefore Delta will not actually switch until it accumulates $SC + rCost - 1$ more cost in $impl_1$ than it would have in $impl_2$. The ϵ term bounds the cost of these boundary conditions.

Examples

In this section we apply the Delta algorithm to two problems: the distributed pub/sub problem and the data structure selection problem.

The distributed pub/sub problem. Consider a data server that serves records from a database to many clients. The server and each client reside on separate nodes of the network. Each client can perform a read or a write on the data. We assume that the clients are independent, that is, a client reads and writes data independent of any other client. There is no synchronization between clients. More formally, for any given "run," there is a linear order in which each client reads and writes records, but the interleaving at the server of reads and writes from different clients is subject to various factors, and cannot be predicted.

Each client can exist in one of two modes: either in *subscription* (Sub) mode or in *nonsubscription* (Non-Sub) mode. In the latter case, for each read that the client wants to perform, it must send a message to the server and receive a reply back. In Sub mode, a client caches a local copy of the database. All reads of the database go against this local copy. In either

case, writes must still go to the server. Upon receiving a write update from any client, the server must inform all subscribers (even the writer of the data if he or she is in Sub mode) of the change to the data. This mechanism is known as "pub/sub" (publication/subscription).

Because it is often impossible to statically predict the read/write behavior of clients, and because their behavior changes over time, we consider an adaptive pub/sub strategy that does not require a client to permanently use either Sub or NonSub mode, but that can flexibly switch between these implementations depending upon current workloads. The goal is to find an optimal strategy for switching implementations that minimizes network traffic.

An instance of the pub/sub problem consists of a server and m clients denoted by $client_j$ $(1 \leq j \leq m)$. Let $r_1, \cdots, r_k$ be a sequence of requests where request r_j is either $read_j$ or $write_j$, $1 \leq j \leq m$, indicating that $client_j$ is either reading or writing a data record to the database. We assume that the database contains $p - 1$ records (by database we simply mean some collection of data items, where each item can be read and written individually). We now focus on how the requests related to $client_i$, $1 \leq i \leq m$, when in NonSub mode, are carried out.

- $read_i(r)$: This request generates a message from $client_i$ to the server asking for record r, and a message back from the server delivering record r.
- $read_j(r), j \neq i$: This request does not concern $client_i$ (it is only relevant to $client_j$).
- $write_i(r)$: This request generates a message from $client_i$ to the server asking for record r to be written.
- $write_j(r), j \neq i$: This request does not concern $client_i$.

The requests related to $client_i$, $1 \leq i \leq m$, when in Sub mode are carried out as follows.

- $read_i(r)$: Record r is read from the cached data at $client_i$, thereby avoiding the need for messages to the server.
- $read_j(r), j \neq i$: This request does not concern $client_i$.
- $write_i(r)$: This request generates a message from $client_i$ to the server asking for record r to be written, and an $update(r)$ message from the server to $client_i$.

Figure 2 Cost to $client_i$ of processing requests $(j \neq i)$

$$
\begin{aligned}
Cost(write_i, NonSub) &= 1\\
Cost(write_j, NonSub) &= 0\\
Cost(read_i, NonSub) &= 2\\
Cost(read_j, NonSub) &= 0\\
SwitchCost(NonSub, Sub) &= p\\
\\
Cost(write_i, Sub) &= 2\\
Cost(write_j, Sub) &= 1\\
Cost(read_i, Sub) &= 0\\
Cost(read_j, Sub) &= 0\\
SwitchCost(Sub, NonSub) &= 1
\end{aligned}
$$

- $write_j(r), j \neq i$: This request generates an $update(r)$ message from the server to $client_i$ informing it that record r has been updated.

Switching implementations is carried out through subscribe and unsubscribe messages.

- *subscribe*( ): This is shorthand for *switch*(*NonSub, Sub*). It generates a message from $client_i$ to the server, subscribing $client_i$ to the database, and a message from the server delivering a local copy of the database to $client_i$. It puts $client_i$ into Sub mode.
- *unsubscribe*( ): This is shorthand for *switch*(*Sub*, *NonSub*). It generates a message from $client_i$ to the server unsubscribing $client_i$. It puts $client_i$ into NonSub mode, and discards its local copy of the database.

The cost of a request is proportional to the number of messages it generates and their size, such that sending a single message containing a single record has unit cost. Figure 2 gives the costs to $client_i$ for each of these requests.

Algorithm Delta can be applied to the pub/sub problem by monitoring each read and write request at $client_i$, switching from NonSub to Sub mode when it detects that Sub mode would have processed a previous subsequence of the requests for $p + 1$ less in network messaging costs than it was processed in NonSub mode. The number $p + 1$ is the switch cost of the Delta algorithm; $SwitchCost(NonSub, Sub) + SwitchCost(Sub, NonSub) = p + 1$. It monitors the requests being processed when in Sub mode and de-

Figure 3 A read/write sequence (α) and a transformed sequence (α')

$\alpha = read_1, write_1, read_1, read_2, read_1, read_1, read_1, write_2, write_1, read_1, read_2, write_2, read_1, read_1, write_2, read_1, read_1$

$\alpha' = subscribe_1, localRead_1, write_1, update_1, localRead_1, read_2, localRead_1, localRead_1, localRead_1, write_2, update_1, write_1, update_1, localRead_1, read_2, write_2, update_1, localRead_1, localRead_1, write_2, update_1, localRead_1, localRead_1$

Figure 4 The processing of sequence α by algorithm Delta

$\alpha = read_1, write_2, write_2, write_2, write_2, read_1, read_1, read_1, write_2, read_1$

Op Number x	0	1	2	3	4	5	6	7	8	9	10
Impl1Cost	0	2	2	2	2	2	4	6	8	8	10
Impl2Cost	0	0	1	2	3	4	4	4	4	5	5
Impl1Cost – Impl2Cost	0	2	1	0	–1	–2	0	2	4	3	5
MinDelta	0	0	0	0	–1	–2	–2	–2	–2	–2	–2
(*Impl1Cost – Impl2Cost*) – *MinDelta*	0	2	1	0	0	0	2	4	6	5	7

cides when to switch to NonSub mode based upon a similar calculation.

Figure 3 gives a sequence α involving two clients and a server. For $client_1$, $Cost(\alpha, NonSub) = 22$ and $Cost(\alpha, Sub) = 7$. If $p + 1 \leq 15$, then Delta would dictate that $client_1$ should subscribe in the course of this sequence. α' is the transformed sequence, if $client_1$ were to subscribe at the beginning of this sequence. In this example, op_i indicates that $client_i$ is performing operation op, where $op \in \{read, localRead, write, subscribe, unsubscribe\}$, $i \in \{1, 2\}$, and $update_i$ indicates that the server is sending an update message to $client_i$. *localRead* indicates a *read* operation from the local cache.

There is one subtlety, however, in using Delta for the pub/sub problem. Looking at Figure 2, we see that Delta needs to know not only the reads and writes done by $client_i$, but also the writes done by any $client_j$, $j \neq i$. If Delta is running at $client_i$, how does it get this information? One possibility is to have Delta run at the server, where all client reads and writes are known, and have the server tell $client_i$ when to switch implementations. However, it may be advantageous to have the control for switching implementations located at the client, thereby providing a more distributed architecture.

In Reference 5 we address the question "How can Delta, running at $client_i$, know about the write operations performed by other clients without increasing the number of messages that must be exchanged between the client and the server?" In Sub mode the issue does not exist, as the client can infer this from the number of updates messages it receives. In NonSub mode, it is required that the server piggyback a count of the total number of write operations it has received on the reply to each read request by $client_i$.

Figure 4 illustrates the processing by Delta of sequence α involving $client_1$, $client_2$, and a server. The value of the counters as well as the intermediate computations at $client_1$ (in NonSub mode) are shown, as calculated by Delta (cf. Figure 1). In Figure 4, $impl_1$ is in NonSub mode while $impl_2$ is in Sub mode.

The adaptive data structure selection problem. Often an application chooses a data structure that is optimized for particular requests. When different types of requests require different data structures, either duplicate data structures are maintained, or a single data structure is used, chosen presumably for the most frequent type of requests. Consider, for example, a set of records S that has two possible keys, i_1 and i_2. (My bank allows me to identify myself either by my social security number or by my account number.) Let *Comp* be the component that encapsulates all operations on S. *Comp* may support many types of requests that operate on this set, but all of these requests can be categorized as either of type r_1, for requests using key i_1, or of type r_2, for requests using key i_2.

One strategy is for *Comp* two create two index tables for S, one using key i_1 and one using key i_2. Each index table is implemented as a dictionary,[6] allowing one to perform the usual operations of finding, inserting, and deleting elements. But this strategy may not always be feasible, because there may not be sufficient capacity for storing both indices (e.g., a pervasive device), or the overhead in maintaining these two separate indices may be too high. In such cases, we can support two implementations, one that uses key i_1, and one that uses key i_2. Algorithm Delta is used to dynamically decide when to switch between these implementations. In Reference 5 we present a more complete discussion on the use of Delta in this context.

Competitiveness

For any two-implementation-component problem, let $SC_1 = SwitchCost(impl_1, impl_2)$, $SC_2 = SwitchCost(impl_2, impl_1)$, and let SC be the round trip switching cost $SC = SC_1 + SC_2$. Given request r let

$$reqCost = max_r|Cost(r, impl_1) - Cost(r, impl_2)|.$$

Let $\epsilon = 2{*}reqCost/SC$. In this section we prove the following theorem:

Theorem 1: *Algorithm Delta is* $(3 + \epsilon)$*-competitive for any two-implementation-component problem.*

Note that when the cost of processing a single request is significantly less than the cost of switching implementations, then $\epsilon \ll 1$ and Delta is close to optimal.

Proof: Consider any sequence α processed by Delta. α can always be viewed as consisting of recurring segments of the form $\omega = \beta, switch(impl_1, impl_2), \gamma, switch(impl_2, impl_1)$, where β and γ are sequences of requests, during processing of β $impl_1$ is active, and during processing of γ $impl_2$ is active. Note that β and γ contain no *switch* statements. Without loss of generality, we can assume that $\beta, \gamma \neq \Omega$ (Ω is the empty sequence). We will maintain the following invariant:

Invariant: At the start of processing of an ω segment, for *any* algorithm processing this sequence, either $impl_1$ is active or $impl_2$ is active and there is a *debt* of SC_1. By "debt" we mean that if, at the start of processing of a segment ω $impl_2$ is active, then a cost SC_1 has accrued in a preceding segment that has not yet been charged to the algorithm. Initially we know this is true because we assume that all algorithms start in $impl_1$ mode.

Let r be the last request in β. Note that $Cost(r, impl_1) \leq Cost(r, impl_2) + reqCost$. There are two properties of note regarding the cost to any algorithm of processing the β portion of segment ω.

Property 1: Let $\mathcal{A}$ be any algorithm that processes β with $impl_1$ active at both the start and the end of the sequence, and switching between implementations any number of times in between. Then $Cost_\beta^{\mathcal{A}} > Cost(\beta, impl_1) - reqCost$; that is, at most *reqCost* is gained by switching back and forth during the processing of β. To see why this is true, consider the case when $\mathcal{A}$ switches only once; say, $\mathcal{A}$ makes $impl_1$ active during β_1, $impl_2$ active during β_2, and $impl_1$ active during β_3, where $\beta = \beta_1, \beta_2, \beta_3$. Assume that $\beta_2 \neq \Omega$, otherwise this is trivially true. If $\beta_3 \neq \Omega$ then $Cost(\beta_2, impl_2) + SC > Cost(\beta_2, impl_1)$, otherwise Delta would have switched to $impl_2$ before or at the end of processing β_2. If $\beta_3 = \Omega$ then let β_2' be the prefix of β_2 with one operation (r) removed from the end of β_2. Then $Cost(\beta_2', impl_2) + SC > Cost(\beta_2', impl_1)$ since Delta did not switch implementations before completing the processing of β. Since $Cost(r, impl_2) \geq Cost(r, impl_1) - reqCost$, we have

$$\begin{aligned} &Cost(\beta_2, impl_2) + SC \\ &\quad = Cost(\beta_2', impl_2) + Cost(r, impl_2) + SC \\ &\quad > Cost(\beta_2', impl_1) + Cost(r, impl_1) - reqCost \\ &\quad = Cost(\beta_2, impl_1) - reqCost \end{aligned}$$

Hence,

$$Cost_{\beta}^{\mathcal{A}} = Cost(\beta_1, impl_1) + Cost(\beta_2, impl_2) + Cost(\beta_3, impl_1) + SC > Cost(\beta, impl_1) - reqCost$$

A similar argument applies to any algorithm that switches multiple times back and forth between implementations during β. This completes the proof of Property 1.

Property 2: For any β_1, β_2 such that $\beta = \beta_1\beta_2$,

$$Cost(\beta, impl_1) < Cost(\beta_1, impl_1) + Cost(\beta_2, impl_2) + SC + reqCost$$

That is, it costs at most $SC + reqCost$ more to process β when $impl_1$ is active than processing part of it when $impl_1$ is active and the rest when $impl_2$ is active. If $\beta_2 = \Omega$, this is trivially true. So assume that $\beta_2 \neq \Omega$ and let β_2' be the prefix of β_2 with one operation (r) removed from the end of β_2. Then

$$Cost(\beta_1\beta_2', impl_1) < Cost(\beta_1, impl_1) + Cost(\beta_2', impl_2) + SC$$

since Delta did not switch implementations before completing the processing of β. Since

$$Cost(r, impl_1) \leq Cost(r, impl_2) + reqCost,$$

we have

$$\begin{aligned} &Cost(\beta, impl_1) \\ &\quad = Cost(\beta_1\beta_2', impl_1) + Cost(r, impl_1) \\ &\quad < Cost(\beta_1, impl_1) + Cost(\beta_2', impl_2) + SC \\ &\qquad + Cost(r, impl_2) + reqCost \\ &\quad = Cost(\beta_1, impl_1) + Cost(\beta_2, impl_2) + SC \\ &\qquad + reqCost \end{aligned}$$

This completes the proof of Property 2.

Let $l_1 = Cost(\beta, impl_1) - SC - reqCost$. Now we determine the cost to any adversary algorithm *Adv* of processing β by examining the four cases A through D.

A: *Adv* starts with $impl_1$ active and ends with $impl_1$ active. If it never switches implementations, its cost is $l_1 + SC + reqCost$. Otherwise, say that it switches from $impl_1$ to $impl_2$ and back to $impl_1$ during β. By Property 1, this can only decrease its cost by *reqCost*, so its cost is at least $l_1 + SC$. Switching multiple times will not further decrease its cost.

B: *Adv* starts with $impl_1$ active and ends with $impl_2$ active. By Property 2, if it switches only once then it may save at most $SC + reqCost$ over the cost of processing β entirely in $impl_1$ but incurs a cost of SC_1 in switching implementations. Hence, its cost is at least $l_1 + SC_1$. Switching multiple times in processing β will not decrease its cost any further.

C: *Adv* starts with $impl_2$ active and ends with $impl_2$ active. If *Adv* does not change modes at all during this phase, then by Property 2 the cost to *Adv* is at least l_1 to process β. However, according to the Invariant, there is a debt of SC_1 to *Adv*, since it starts β with $impl_2$ active. We remove that debt and charge it to this phase of the algorithm. Hence, *Adv* will have at least the cost of $l_1 + SC_1$. By an argument similar to Property 1, by changing its mode during the processing of this sequence, *Adv* will not decrease its cost any further.

D: *Adv* starts with $impl_2$ active and ends with $impl_1$ active. This is similar to the preceding case, except that it has an additional cost of SC_2 for switching to $impl_1$. Hence, its cost is at least $l_1 + SC_1 + SC_2 = l_1 + SC$.

Now consider γ. Let r' be the last request in γ. Note that

$$Cost(r', impl_2) \leq Cost(r', impl_1) + reqCost$$

The two properties stated above for β have analogs in processing γ (we omit the proofs as they are similar to those given above).

Property 3: Let $\mathcal{A}$ be any algorithm that processes γ by initially starting with $impl_2$ active, ending with $impl_2$ active, but switching between implementations any number of times in between. Then $Cost_{\gamma}^{\mathcal{A}} > Cost(\gamma, impl_2) - reqCost$.

Property 4: For any γ_1, γ_2 such that $\gamma = \gamma_1\gamma_2$,

$$Cost(\gamma, impl_2) < Cost(\gamma_1, impl_2) + Cost(\gamma_2, impl_1) + SC + reqCost$$

Let $l_2 = Cost(\gamma, impl_2) - SC - reqCost$. Now we use Properties 1 through 4 above and determine the

cost to any adversary *Adv* of processing γ by examining the four cases E through H.

E: *Adv* starts with $impl_2$ active and ends with $impl_2$ active. If it never switches implementations, its cost is $l_2 + SC + reqCost$. Otherwise, say, it switches from $impl_2$ to $impl_1$ and back to $impl_2$ during processing of γ. By Property 3, this can decrease its cost by at most *reqCost*. However, since *Adv* ends the processing of segment ω with $impl_2$ active (and will start the next segment with $impl_2$ active), we remove SC_1 in cost from this phase of the algorithm and pay the debt, thereby maintaining the Invariant. Its costs are therefore $l_2 + SC - SC_1 = l_2 + SC_2$.

F: *Adv* starts with $impl_2$ active and ends with $impl_1$ active. By Property 4, if it only switches once, then it may save at most $SC + reqCost$ over processing γ entirely with $impl_2$ active but incurs a cost of SC_2 to switch modes. Hence, its cost is at least $l_2 + SC_2$. Switching multiple times in processing γ will not decrease its cost any further.

G: *Adv* starts in $impl_1$ mode and ends in $impl_1$ mode. If *Adv* does not change modes at all during this phase, then by Property 4 the cost to *Adv* is at least l_2 to process γ. By an argument similar to Property 3, by changing its mode during the processing of this sequence, *Adv* will not decrease its cost any further.

H: *Adv* starts in $impl_1$ mode and ends in $impl_2$ mode. This is similar to the preceding case, except that it has an additional cost of SC_1 for switching to $impl_2$. However, since *Adv* ends the iteration of the ω segment in $impl_2$, we need to remove SC_1 in cost from this phase of the algorithm and assign it to the debt in order to maintain the Invariant. Hence, its cost is at least l_2.

To analyze the overall complexity of *Adv*, we consider the cost of executing one complete segment of ω. *Adv* must follow one of the following eight "paths": $A \rightarrow G$, $A \rightarrow H$, $B \rightarrow E$, $B \rightarrow F$, $C \rightarrow E$, $C \rightarrow F$, $D \rightarrow G$, $D \rightarrow H$. For instance, the first path says that *Adv* first implements case A for β and then implements case G for γ. Note that these eight paths are the only ones possible, as A cannot be combined with E, for example, since case A states that $impl_1$ is active at the end of β, and case E states that $impl_2$ is active at the start of γ (= end of β). By summing the costs of each of these paths, one sees that $Cost_{\beta\gamma}^{Adv} \geq l_1 + l_2 + SC$. The cost to Delta is at most $l_1 + SC + reqCost$ to process β, SC_1 to switch to $impl_2$ at the end of β, $l_2 + SC + reqCost$ to process γ, and SC_2 to switch to $impl_1$ at the end of γ. Hence,

$$\begin{aligned} Cost_{\beta\gamma}^{Delta} &\leq l_1 + SC + reqCost + l_2 + SC \\ &\quad + reqCost + SC_1 + SC_2 \\ &= l_1 + l_2 + 3*SC + 2*reqCost \\ &\leq (3 + \epsilon)*(l_1 + l_2 + SC) \\ &\leq (3 + \epsilon)*Cost_{\beta\gamma}^{Adv} \end{aligned}$$

Hence, Delta is $(3 + \epsilon)$-competitive on any ω segment.

To complete the proof of this theorem, we need to show that for any arbitrary long sequence that is a strict subsequence of an ω segment, Delta is also $(3 + \epsilon)$-competitive. We leave this as a straightforward exercise, based on the discussion above.

End of proof

Lower bounds

In this section we show that there are instances of the two-implementation-component problem such that no algorithm is better than 3-competitive. The proof involves the dynamic pub/sub problem, given in the section "Examples."

In analyzing adaptive on-line algorithms, it is useful to differentiate between different types of adversaries.[4,7] An *oblivious* adversary is one that constructs the sequence of requests without regard to the actions taken by the algorithm. An *adaptive* adversary determines the action corresponding to each element of a sequence on line by taking into account the previous choices made by algorithm. If, additionally, the adversary processes the sequence only after the entire sequence has been generated, it is then said to be an *adaptive off-line* adversary. This is the most powerful sort of adversary, and we follow the terminology of Reference 8 and call it a *strong* adversary.

The result of this section shows that no deterministic or randomized algorithm can do better than Delta against a strong adversary for all two-implementation-component problems. Also, since the distinction between strong and oblivious adversaries is irrelevant for deterministic algorithms,[7] it also shows that no deterministic algorithm is better than 3-competitive against any type of adversary for *all* two-implementation-component problems.

Lemma 1: *Let Alg be any adaptive pub/sub algorithm controlling an individual client. There exists an arbitrarily long sequence α constructed by a strong adversary such that, as the length of α tends to infinity, Alg is at best 3-competitive on α.*

Proof: Consider just two clients, $client_1$ and $client_2$, and let $client_1$ be running algorithm *Alg*. For any integer l, we can construct a sequence of the form $\alpha = read_1^{m_1}\ write_2^{n_1}\ read_1^{m_2}\ write_2^{n_2} \cdots read_1^{m_k}\ write_2^{n_k}$ with the following characteristics (r_1^m indicates m consecutive $client_1$ requests of type r):

- The length of the sequence (number of operations) is larger or equal to l
- For all t, $1 \leq t \leq k$, $client_1$ subscribes after the $read_1^{m_t}$ operations and unsubscribes after the $write_2^{n_t}$ operations.

To find this sequence, we begin by "serving" read requests to the $client_1$ until it subscribes and then "serving" write requests to $client_2$, causing update messages to propagate to $client_1$ until $client_1$ unsubscribes. We continue this process until we reach the desired sequence length. Since we are dealing with a strong adversary, it knows exactly when to serve read (write) requests, as it just waits until it sees $client_1$ subscribe (unsubscribe). We assume that $client_1$ forever alternates between Sub and NonSub modes if served enough write and read requests, respectively. Otherwise, this client would trivially not be competitive. Indeed, if it stayed in Sub mode, the adversary could forever feed it update messages whereas the adversary would incur no cost by staying in NonSub mode. Similarly, if it stayed in NonSub mode the adversary could forever feed it read messages and the adversary would incur no cost by staying in Sub mode.

Let R be the total number of read operations and W be the total number of write operations in α. The cost to $client_1$ incurred by *Alg*, which we denote by $Cost_\alpha^{Alg}(1)$, equals $2R + W + (pk + k)$, as it costs $2R$ for $client_1$ read messages, W for $client_1$ update messages ($= client_2$ write operations), pk for the k $client_1$ subscribe messages, and k for the k $client_1$ unsubscribe messages. Let $M = min\{2R, W, (pk + k)\}$. Now devise the strong adversary $\mathcal{SA}$ as follows:

- Case 1: $M = 2R$. Let $\mathcal{SA}$ stay in NonSub mode throughout the sequence. Then $Cost_\alpha^{\mathcal{SA}}(1) = 2R$. $Cost_\alpha^{Alg}(1) = 2R + W + (pk + k) \geq 3(2R) = 3 * Cost_\alpha^{\mathcal{SA}}(1)$. Hence, *Alg* is at best 3-competitive.
- Case 2: $M = pk + k$. Let $\mathcal{SA}$ subscribe directly before each set of read operations and unsubscribe directly before each set of write operations. Then $Cost_\alpha^{\mathcal{SA}}(1) = pk + k$. $Cost_\alpha^{Alg}(1) = 2R + W + (pK + k) \geq 3(pk + k) = 3 * Cost_\alpha^{\mathcal{SA}}(1)$. Hence, *Alg* is at best 3-competitive.
- Case 3: $M = W$. Let $\mathcal{SA}$ initially subscribe and then stay in Sub mode throughout the sequence. Then $Cost_\alpha^{\mathcal{SA}}(1) = W + p$. Recall that as $l \to \infty$, we can assume $W \to \infty$, otherwise the client is trivially not competitive. Hence, as $l \to \infty$, $Cost_\alpha^{\mathcal{SA}}(1) \to W$. Therefore $Cost_\alpha^{Alg}(1) = 2R + W + (pK + k) \geq 3W = 3 * Cost_\alpha^{\mathcal{SA}}(1)$, and *Alg* is at best 3-competitive.

End of proof

Related work

The ideas in this paper are related to prior work in program optimization, competitive algorithms, and data replication policies.

Some of the work in program optimization discusses optimizing programs by choosing among a set of alternate implementations. For the most part, this work focuses on low-level static optimizations. For example, the early work on SETL examined how to choose the best set representation based upon program analysis.[9] Similarly the work on SQL (Structured Query Language) optimization can be viewed as statically choosing the best implementation for retrieving information from a database given a particular query.[10]

Closer to our strategy is the recent work described in Reference 11, in which the authors propose that the component writer provide multiple implementations of a component, assign costs to the different operations on each component, profile a program made up of multiple components running in a particular context, and construct an object affinity graph for this program. They then use a graph partitioning algorithm to find an optimal partitioning of the graph, thereby finding the optimal implementation for each component. Central to their scheme is the observation that the implementations at different components affect each other. Unlike algorithm Delta, their strategy optimizes a program globally, not just at the level of an individual component. On the other hand, their technique requires a more complicated methodology (profiling and the creation of an object affinity graph) and cannot adjust to dynamically changing contexts as Delta does.

Recent work on dynamic architecture description languages (see Reference 12 for a summary) aims at building software architectures that can respond adaptively to changes. Algorithms like Delta will be important in helping realize the goals of these architectures.

Our adaptive pub/sub application contrasts with previous work on data replication.[13,14] An extended version of Reference 15 gives a good overview of the literature on file allocation, file migration, and file replication problems. Although similar in some ways, our adaptive pub/sub application is closer to the work in References 16 and 17, which we discuss later in this section.

Our work draws inspiration from many competitive algorithms.[4,8,15,18,19] Although it may seem that this work could be used to solve our problem, a closer examination shows that this is not the case. For instance, one immediate idea for a competitive adaptive pub/sub algorithm, modeled after the snoopy caching algorithm,[8,18] is for a client to subscribe after performing $(1/2)p$ consecutive read operations, and to unsubscribe after receiving p consecutive update operations. The asymmetry between reads and updates is due to the fact that in our model a read is twice as expensive as an update operation. Consider the sequence $(read_1^{(1/2)p-1}, write_2)^n$; that is, client$_1$ issues $(1/2)p - 1$ reads followed by a write by *client*$_2$. This is repeated n times. The cost to the snoopy caching algorithm for this sequence with regard to client$_1$ would be $n * (p - 2)$. One can devise an *adversary* that would process this sequence at cost $p + n$ (the adversary would initially subscribe and then stay in Sub mode). There do not exist constants c and d *independent* of p such that $n(p - 2) \leq c(n + p) + d$ for all n. Hence, the snoopy caching algorithm is not a good competitive algorithm for the pub/sub problem.

As mentioned, our work on the adaptive pub/sub problem is most closely related to the work in References 16 and 17. Reference 16 seems to exactly address our problem and to derive a better lower bound than ours ($2k$-competitive, for any integer k). In their model, however, the data being replicated are always of unit size, that is, variable $p - 1$ (the size of the data being replicated) always takes value 1. Hence, our work generalizes that result in an important way, by allowing arbitrary size data.

Let us see how the algorithm in Reference 16 would work for our problem. This algorithm re-evaluates the mode a client is in after every k operations (alternatively, the algorithm in Reference 20 re-evaluates after a fixed time period t). If read operations are the majority, the client switches to (stays in) Sub mode; otherwise it switches to (stays in) NonSub mode. Consider the sequence $(read_1^k, write_2^k)^n$; that is, k reads by client$_1$ are followed by k writes by some other client. This is repeated n times. The cost with respect to client$_1$ would be $(p + 1)n + 3kn$. One can devise an adversary whose cost would be $W = min\{nk, (p + 1)n\}$. There do not exist constants c and d independent of p such that $(p + 1)n + 3kn \leq cW + d$ for all n. Because p may be arbitrarily large, this may result in a large competitive factor. Hence, the algorithm would not perform well in the worst case when one generalizes to larger data structures. The algorithm does model other aspects that we have not considered. Most importantly, whereas we assume a fixed data "server," Reference 16 uses the notion of a *primary site* that can dynamically change in the course of execution.

Dynamically switching between implementations of a component should become a useful tool in autonomic computing.

Very recently, as this paper was going into production, I became aware of the very relevant work on metrical task systems.[21] Metrical task systems are very similar to the adaptive component problem described in this paper, and the results there are consistent with the results of this paper. However, there are two important differences between the two. First, in metrical task systems, it is assumed that $SwitchCost(i, j) = SwitchCost(j, i)$ (symmetry). We make no such assumption. As a matter of fact, for the distributed pub/sub problem described in the section "Examples," this is not the case. Second, in our model we assume that once a request is received, the processing of the request must be completed before switching to another implementation. We took this approach to better model scenarios where rapid response time is critical (i.e., the response must be made before embarking on the relatively lengthy process of switching implementations). In metrical task systems, however, switching implementations (states) can be done after a request is received but before it is processed. For both of these reasons, our work

generalizes metrical task systems in the case of two implementations (or, in the terminology of Reference 21, in the case of two states), and it is not clear that the algorithms and proofs given in metrical task systems apply directly to the adaptive component problem.

Summary and open issues

This paper provides a framework for analyzing the effectiveness of components that switch implementations dynamically at run time based upon run-time workload characteristics. We believe that dynamic selection of component implementations will become an effective tool in optimizing component performance, and therefore the framework given in this paper should become increasingly important.

An important contribution of this paper is the near optimal $(3 + \epsilon)$-competitive algorithm Delta, for the *two-implementation-component problem*, in which there are only two implementations to choose from. An obvious question is whether there exist competitive algorithms for components with an arbitrary number of implementations. In Reference 5 we show there does not exist an algorithm that is better than k-competitive when the number of implementations is k. Another limitation of Delta is that it requires a switch from one implementation to another in "one shot," and this can often be expensive. In Reference 5 we show how Delta can switch implementations in an incremental fashion for the two example problems of section "Examples," thereby amortizing the cost of the switch operation. Still, a more general method of incremental switching between component implementations is needed.

Competitiveness is only one criterion by which to judge an algorithm for switching among implementations. In Reference 5 we mention one other criterion: convergence. Other metrics can be used as well. For instance, it may be that some domains exhibit great periodicity of patterns in event sequences. The ability to learn these patterns and then optimize to them is an important strategy in these domains.

Most importantly, we need experience to see how well Delta works in practice. This will shed light on how to best evolve this algorithm. For instance, it may be that domain knowledge needs to be taken into account in order to obtain better results.

Cited references

1. E. Roman, S. W. Ambler, T. Jewell, and F. Marinescu, *Mastering Enterprise JavaBeans*, *2nd Edition*, John Wiley & Sons, Inc., Hoboken, NJ (December 2001).
2. S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama, *Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI*, Sams Publishing, Indianapolis, IN (December 2001).
3. D. D. Sleator and R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM* **28**, No. 2, 202–208 (February 1985).
4. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, UK (1995).
5. D. Yellin, "Competitive Algorithms for the Dynamic Selection of Component Implementations," unpublished. Available from the author at dmy@us.ibm.com.
6. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, MA (1974).
7. S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson, "On the Power of Randomization in Online Algorithms," *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM, New York (1990), pp. 379–386.
8. A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, "Competitive Randomized Algorithms for Non-Uniform Problems," *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM, New York (January 1990), pp. 301–309.
9. J. Schwartz, "Automatic Data Structure Choice in a Language of Very High Level," *Communications of ACM* **18**, No. 12, 722–728 (1975).
10. J. Ullman, *Principles of Database and Knowledge-Base Systems*, Volume 2, Computer Science Press, Rockville, MD (1989).
11. K. Hogstedt, D. Kimelman, V. T. Rajan, T. Roth, V. Sreedhar, M. Wegman, and N. Wang, "The Autonomic Performance Prescription," unpublished. Available from the author at wegman@us.ibm.com.
12. N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering* **26**, No. 1, 70–93 (January 2000).
13. R. Ladin, B. Liskov, and L. Shrira, "A Technique for Constructing Highly Available Distributed Services," *Algorithmica* **3**, 393–420 (1988).
14. R. Alonso, D. Barbara, and H. Garcia Molina, "Data Caching Issues in an Information Retrieval System," *ACM Transactions on Database Systems* **15**, No. 3, 359–384 (1990).
15. Y. Bartal, A. Fiat, and Y. Rabani, "Competitive Algorithms for Distributed Data Management," *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing* (STOC), ACM, New York (May 1992), pp. 39–50.
16. Y. Huang and O. Wolfson, "A Competitive Dynamic Data Replication Algorithm," *Proceedings of the Ninth International Conference on Data Engineering* (ICDE93), Vienna, Austria, April 1993; IEEE, New York (1993), pp. 310–317.
17. O. Wolfson and S. Jajodia, "An Algorithm for Dynamic Data Distribution," *Proceedings of the 2nd Workshop on the Management of Replicated Data* (WMRD-II), IEEE, New York (November 1992), pp. 62–65.
18. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. Sleator, "Competitive Snoopy Caching," *Algorithmica* **3**, No. 1, 70–119 (1988).

19. A. Fiat, R. Karp, M. Luby, L. A. McGeoch, D. Sleator, and N. E. Yong, "Competitive Paging Algorithms," *Journal of Algorithms* **12**, No. 4, 685–699 (1991).
20. O. Wolfson, S. Jajodia, and Y. Huang, "An Adaptive Data Replication Algorithm," *ACM Transactions on Database Systems* (TODS) **22**, No. 2, 255–314 (1997).
21. A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, UK (1998). See Chapter 9 "Metrical Task Systems."

Accepted for publication August 30, 2002.

Daniel M. Yellin *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (dmy@us.ibm.com).* Dr. Yellin joined the Watson Research Center in 1987 after completing a Ph.D. program in computer science at Columbia University. He has authored over 20 journal and conference papers, as well as two computer science books, in diverse areas, which include distributed computing, program analysis, and algorithms for incremental computations. Dr. Yellin has held various positions on research and standards committees. He was editor of the International Organization for Standardization (ISO) standard on Remote Procedure Call and Vice-Chair of Operations for the Association of Computing Machinery (ACM) Special Interest Group on Programming Languages (SIGPLAN). In April 1999 he was named IBM Distinguished Engineer, and in August 1999 he was elected to the IBM Academy of Technology. He currently is Director, Software Technology Department, at IBM Research.