Integrating Remote Invocation and Distributed Shared State*

Chunqiang Tang, DeQing Chen,
Sandhya Dwarkadas, and Michael L. Scott

Computer Science Department, University of Rochester
{sarrmor,lukechen,sandhya,scott}@cs.rochester.edu

Abstract

Most distributed applications require, at least conceptually, some sort of shared state*: information that is non-static but mostly read, and needed at more than one site. At the same time, RPC-based systems such as Sun RPC, Java RMI, CORBA, and .NET have become the* de facto *standards by which distributed applications communicate. As a result, shared state tends to be implemented either through the redundant transmission of deep-copy RPC parameters or through ad-hoc, application-specific caching and coherence protocols. The former option can waste large amounts of bandwidth; the latter significantly complicates program design and maintenance.*

To overcome these problems, we propose a distributed middleware system that works seamlessly with RPC-based systems, providing them with a global, persistent store that can be accessed using ordinary reads and writes. Relaxed coherence models and aggressive protocol optimizations reduce the bandwidth required to maintain shared state. Integrated support for transactions allows a chain of RPC calls to update shared state atomically.

We focus in this paper on the implementation challenges involved in combining RPC with shared state and transactions. In particular, we describe a transaction metadata table *that allows processes inside a transaction to share data invisible to other processes and to exchange data modifications efficiently. Using microbenchmarks and a large-scale datamining application, we demonstrate how the integration of RPC, transactions, and shared state facilitates the rapid development of robust, maintainable code.*

1. Introduction

Most Internet-level applications are distributed not for the sake of parallel speedup, but rather to access people, data, and devices in geographically disparate locations. Typical examples include e-commerce, computer-supported-collaborative work, multi-player games, peer-to-peer data sharing, and scientific GRID computing. For the sake of availability, scalability, latency, and fault tolerance, most such applications cache information at multiple sites. To maintain these copies in the face of distributed updates, programmers typically resort to ad-hoc messaging or RPC protocols that embody the coherence and consistency requirements of the application at hand. The code devoted to these protocols often accounts for a significant fraction of overall application size and complexity, and this fraction is likely to increase.

To facilitate the design, implementation, maintenance, and tuning of distributed applications, we have developed a system known as InterWeave that manages shared state automatically [4, 20, 22]. InterWeave allows the programmer to share both statically and dynamically allocated variables across programs written in different programming languages and running on a wide variety of hardware and OS platforms. InterWeave currently supports C, C++, Java, Fortran 77, and Fortran 90, running on Alpha, Sparc, x86, MIPS, and Power series processors, under Tru64, Solaris, Linux, Irix, AIX, and Windows NT (XP). Driving applications include datamining, intelligent distributed environments, and scientific visualization.

Shared data segments in InterWeave are named by URLs, but are accessed, once mapped, with ordinary loads and stores. Segments are also persistent, outliving individual executions of sharing applications, and support a variety of built-in and user-defined coherence and consistency models. Aggressive protocol optimizations, embodied in the InterWeave library, allow InterWeave applications to outperform all but the most sophisticated examples of ad-hoc caching [4, 5, 22].

Most distributed applications, despite their need for shared state, currently use remote invocation to transfer control among machines. InterWeave is therefore designed to be entirely compatible with RPC systems such as Sun RPC, Java RMI, CORBA, and .NET. By specifying where computation should occur, RPC allows an application to balance load, maximize locality, and co-locate computation

*This work was supported in part by NSF grants CCR-9988361, CCR-0204344, CCR-0219848, ECS-0225413, and EIA-0080124; by DARPA/ITO under AFRL contract F29601-00-K-0182; by the U.S. Dept. of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460; and by equipment or financial grants from Compaq, IBM, Intel, and Sun.

with devices, people, or private data at specific locations. At the same time, shared state serves to

- eliminate invocations devoted to maintaining the coherence and consistency of cached data;
- support genuine reference parameters in RPC calls, eliminating the need to pass large structures repeatedly by value, or to recursively expand pointer-rich data structures using deep-copy parameter modes;
- reduce the number of trivial invocations used simply to put or get data.

These observations are not new. Systems such as Emerald [12] and Amber [3] have long employed shared state in support of remote invocation in homogeneous object-oriented systems. Kono et al. [13] supported reference parameters and caching of remote data in a heterogeneous environment, but with a restricted type system, and with no provision for coherence across RPC sessions.

Remote invocation mechanisms have long supported automatic deep-copy transmission of structured data among heterogeneous languages and machine architectures [11, 25], and modern standards such as XML provide a language-independent notation for structured data. To the best of our knowledge, however, InterWeave is the first system to automate the typesafe sharing of structured data in its *internal* (in-memory) form across multiple languages and platforms, and to optimize that sharing for distributed applications.

For Internet-level applications, system failures or race conditions are common when accessing shared data. Since fault tolerance is not provided at the RPC level, RPC-based applications usually have to build their own mechanism to recover from faults and to improve availability. Recovery is particularly tricky in the face of cached shared state. InterWeave eases the task of building robust distributed applications by providing them with support for transactions. A sequence of RPC calls and data access to shared state can be encapsulated in a transaction in such a way that either all of them execute or none of them do with respect to the shared state. Transactions also provide a framework in which the body of a remote procedure can see (and optionally contribute to) shared data updates that are visible to the caller but not yet to other processes.

Previous papers have focused on InterWeave's coherence and consistency models [4], heterogeneity mechanisms [22], and protocol optimizations [5]. The current paper addresses the integration of shared state, remote invocation, and transactions. Section 2 briefly summarizes the InterWeave programming model, and introduces a design that seamlessly integrates shared state, remote invocation, and transactions to form a distributed computing environment. Section 3 sketches the basic structure of the InterWeave client library and server, and then focuses on implementation issues in adding support for remote invocation and transactions into InterWeave. Section 4 evaluates InterWeave in both local and wide area networks, using microbenchmarks and a larger example, drawn from our work in data mining, that uses RPC and shared state to offload computations to back-end servers. Section 5 discusses related work. Section 6 presents conclusions.

2. InterWeave Design

InterWeave integrates shared state, remote invocation, and transactions into a distributed computing environment. The InterWeave programming model assumes a distributed collection of servers and clients. Servers maintain persistent copies of shared data and coordinate sharing among clients. Clients in turn must be linked with a special InterWeave library, which arranges to map a cached copy of needed data into local memory. Once mapped, shared data (including references) are accessed using ordinary reads and writes. InterWeave servers are oblivious to the programming languages used by clients, and the client libraries may be different for different programming languages. InterWeave supports the use of relaxed coherence models when accessing shared data. Updates to shared data and invocations to remote procedures on arbitrary InterWeave processes can be optionally protected by transactions.

2.1. Data Allocation

The unit of sharing in InterWeave is a self-descriptive *segment* (a heap) within which programs allocate strongly typed *blocks* of memory. Every segment is specified by an Internet URL. The blocks within a segment are numbered and optionally named. By concatenating the segment URL with a block name or number and optional offset (delimited by pound signs), we obtain a *machine-independent pointer (MIP)*: "`foo.org/path#block#offset`". To accommodate heterogeneous data formats, offsets are measured in primitive data units—characters, integers, floats, etc.—rather than in bytes.

Every segment is managed by an InterWeave server at the IP address corresponding to the segment's URL. Different segments may be managed by different servers. Assuming appropriate access rights, `IW_open_segment()` communicates with the appropriate server to open an existing segment or to create a new one if the segment does not yet exist. The call returns an opaque segment handle, which can be passed as the initial argument in calls to `IW_malloc()`.

As in multi-language RPC systems, the types of shared data in InterWeave must be declared in an interface description language (IDL—currently Sun XDR). The InterWeave IDL compiler translates these declarations into the appropriate programming language(s) (C, C++, Java, Fortran). It also creates initialized *type descriptors* that specify the layout of the types on the specified machine. The descriptors must be registered with the InterWeave library prior to being used, and are passed as the second argument in calls to `IW_malloc()`. These conventions allow the library to translate data to and from wire format, ensuring that each type will have the appropriate machine-specific byte order, alignment, etc. in locally cached copies of segments.

Given a pointer to a block in an InterWeave segment, or to data within such a block, a process can create a corresponding MIP: "`IW_mip_t m = IW_ptr_to_mip(p)`". This MIP can then be passed to another process through a parameter of a remote procedure. Given appropriate access rights, the other process can convert it back to a machine-specific pointer, as in "`my_type *p = (my_type*) IW_mip_to_ptr(m)`". The `IW_mip_to_ptr()` call reserves space for the specified segment if it is not cached locally, and returns a local machine address. Actual data for the segment will not be copied into the local machine unless and until the segment is locked.

2.2. Coherence

Synchronization takes the form of reader-writer locks that take a segment handle and a transaction handle as parameters. A process must hold a writer lock on a segment in order to allocate, free, or modify blocks. When modified by clients, InterWeave segments move over time through a series of internally consistent states. The server for a segment need only maintain a copy of the segment's most recent version; clients cache entire segments, so they never need a "missing piece" of something old.

When a process first locks a shared segment (for either read or write), the InterWeave library obtains a copy from the segment's server. At each subsequent read-lock acquisition, the library checks to see whether the local copy of the segment is "recent enough" to use [4]. If not, it obtains an update from the server. Twin and diff operations [2], extended to accommodate heterogeneous data formats [22], allow the implementation to perform an update in time proportional to the fraction of the data that has changed.

2.3. RPC and Transactions

InterWeave's shared state can be used with RPC systems by passing MIPs as ordinary RPC string arguments. When necessary, a sequence of RPC calls, lock operations, and data manipulations can be protected by a transaction to ensure that distributed shared state is updated atomically. Operations in a transaction are performed in such a way that either all of them execute or none of them do with respect to InterWeave shared state. InterWeave may run transactions in parallel, but the behavior of the system is equivalent to some serial execution of the transactions, giving the appearance that one transaction runs to completion before the next one starts (more information on relaxed transactions can be found in the TR version of this paper [23]). Once a transaction commits, its changes to the shared state survive failures.

A transaction starts with an `IW_begin_work()` call, which returns an opaque transaction handle to be used in later transactional operations, such as `IW_commit_work()` and `IW_rollback_work()`. Each RPC call automatically starts a sub-transaction that can be individually aborted without rolling back the work that has been done by outer (sub-)transactions. In keeping with traditional RPC semantics, we assume that only one process in an RPC call chain is active at any given time.

The skeleton code for the RPC client and server is generated using the standard `rpcgen` tool, then slightly modified by the InterWeave IDL compiler to insert a transaction handle field in both the RPC argument and result structures. The XDR translation routines for the arguments and results are also augmented with a call to `xdr_trans_arg()` or `xdr_trans_result()`, respectively. These two InterWeave library functions encode and transmit transaction information along with other RPC arguments or results.

An RPC caller can pass references to shared state (MIPs) to the callee as ordinary string arguments. The RPC callee then locks the segment and operates on it. The callee can see shared data updates that are visible to the caller but not yet to other processes. Modifications to the segment made by the callee will be visible to other processes in the transaction when the lock is released, and will be applied to the InterWeave server's master copy when the outermost (root) transaction commits. Before the root transaction commits, those modifications are invisible to other transactions.

In addition to providing protection against various system failures, transactions also allow applications to recover from problems arising from relaxed coherence models, e.g., deadlock or lock failure caused by inter-segment inconsistency. Suppose, for example, that process P has acquired a reader lock on segment A, and that the InterWeave library determined at the time of the acquire that the currently cached copy of A, though not completely up-to-date, was "recent enough" to use. Suppose then that P attempts to acquire a lock on segment B, which is not yet locally cached. The library will contact B's server to obtain a current copy. If that copy was created using information from a more recent version of A than the one currently in use at P, a consistency violation has occurred. Users can disable this consistency check if they know it is safe to do so, but under normal circumstances the attempt to lock B must fail. The problem is exacerbated by the fact that the information required to track consistency (which segment versions depend on which?) is unbounded. InterWeave hashes this information in a way that is guaranteed to catch all true consistency violations, but introduces the possibility of spurious apparent violations [4]. Transaction aborts and retries can be used in this case to recover from inconsistency, with automatic undo of uncommitted segment updates. An immediate retry is likely to succeed, because P's out-of-date copy of A will have been invalidated.

3. Implementation of InterWeave

In this section, we first sketch the basic structure of the InterWeave client library and server. We then elaborate on the support for RPC and transactions. Details of the basic implementation can be found in previous papers. InterWeave currently consists of approximately 45,000 lines of heavily commented C++ code.

3.1. Basic Implementation

When a client acquires a writer lock on a given segment, the InterWeave library asks the operating system to disable write access to the pages that comprise the local copy of the segment. When a write fault occurs, the `SIGSEGV` signal handler, installed by the InterWeave library at program startup time, creates a pristine copy, or *twin* [2], of the page in which the write fault occurred. It saves a pointer to that twin for future reference, and then asks the operating system to re-enable write access to the page.

When a process releases a writer lock, the library gathers local changes, converts them into machine-independent wire format in a process called *diff collection*, and sends the diff to the server. The changes are expressed in terms of segments, blocks, and offsets of primitive data units (integers, doubles, chars, etc.), rather than pages and bytes. The diffing routine must have access to type descriptors (generated automatically by the InterWeave IDL compiler) in order to compensate for local byte order, word size, and alignment, and in order to swizzle pointers. The content of each descriptor specifies the substructure and layout of its type.

Each server maintains an up-to-date copy of each of the segments for which it is responsible, and controls access to those segments. Upon receiving a diff from a client, an InterWeave server uses the diff to update its master copy.

When a client acquires a reader lock and determines that its local cached copy of the segment is not recent enough to use under the desired coherence model (communicating with the server to make the decision if necessary [4]), the client asks the server to build a wire-format diff that describes the data that have changed between the current local copy at the client and the master copy at the server.

When the diff arrives the library uses it to update the local copy in a process called *diff application*. In the inverse of diff collection, the diff application routine uses type descriptors to identify the local-format bytes that correspond to primitive data changes in the wire-format diff.

3.2. Support for RPC and Transactions

When neither transactions nor RPC are being used, segment diffs sent from an InterWeave client to a server are immediately applied to the server's master copy of the segment. With transactions, updates to the segment master copy are deferred until the transaction commits. Like many database systems, InterWeave employs a strict two-phase locking protocol and two-phase commit protocol to support atomic, consistent, isolated, and durable (ACID) transactions. With a strict two-phase locking protocol, locks acquired in a transaction or sub-transaction are not released to the InterWeave server until the outermost (root) transaction commits or aborts.

Each InterWeave client runs a transaction manager (TM) thread that keeps tracks of all on-going transactions involving the given client and listens on a specific TCP port for transaction related requests.

In the `IW_start_work()` call, a *transaction metadata table* (TMT) is created to record information about the new transaction: locks acquired, locks currently held, version numbers of locked segments, segments modified, locations where diffs can be found, etc. The TMT is the key data structure that supports the efficient implementation of transactions and the integration of shared state and transactions with RPC. It is passed between caller and callee in every RPC call and return. With the aid of the TMT, processes cooperate inside a transaction to share data invisible to other processes and to exchange data modifications without the overhead of going through the InterWeave server. This direct exchange of information is not typically supported by database transactions, but is crucial to RPC performance.

Locks inside a Transaction

When a client requests a lock on a segment using either `IW_twl_acquire()` (for a writer lock) or `IW_trl_acquire()` (for a reader lock), the InterWeave library searches the TMT to see if the transaction has already acquired the requested lock. There are four possible cases. (1) The lock is found in the TMT but another process in the transaction is currently holding an incompatible lock on the segment (e.g., both are write locks). This is a synchronization error in the application. The transaction aborts. (2) The lock is found in the TMT and no other process in the transaction is currently holding an incompatible lock on the segment. The lock request is granted locally. (3) The lock is found in the TMT but only for reading, and the current request is for writing. The client contacts the InterWeave server to upgrade the lock. (4) The lock is not found in the TMT, meaning that the segment has not previously been locked by this transaction. The client contacts the InterWeave server to acquire the lock and updates the TMT accordingly.

When a client releases a lock, the InterWeave library updates the lock status in the TMT. In keeping with the strict two-phase locking semantics, the transaction retains the lock until it commits or aborts rather than returning the lock to the InterWeave server immediately. During the release of a write lock, the library collects a diff that describes the modifications made during the lock critical section. Unlike the non-transaction environment where the diff is sent to the InterWeave server immediately, the diff is stored locally in the *created-diff buffer* (or in a file, if memory is scarce). The library also increases the segment's current version number, stores this number in the TMT, and appends an entry indicating that a diff that upgrades the segment to this new version has been created by this client. The actual content of the diff is not stored in the TMT.

Interplay of RPC and Transactions

When a client performs an RPC inside a transaction, the `xdr_trans_arg()` call, included in the argument marshaling routine by the InterWeave IDL compiler, encodes and transmits the TMT to the callee along with other arguments. A complementary `xdr_trans_arg()` call on the callee side will reconstruct the TMT when unmarshaling the arguments. Typically the TMT is small enough to have a negligible impact on the overhead of the call. For instance, a complete TMT containing information about a single segment is only 76 bytes in length. A null RPC call over a 1Gbps network takes 0.212ms, while a null RPC call in InterWeave (with this TMT) takes just 0.214ms.

Among other things, the TMT tracks the latest version of each segment ever locked in the transaction. This latest version can be either the InterWeave server's master copy or a tentative version created in the on-going transaction. When the callee acquires a lock on a segment and finds that it needs an update (by comparing the latest version in the TMT to the version it has cached), it consults the TMT to decide whether to obtain diffs from InterWeave servers, from other InterWeave clients, or both. To fetch diffs from other clients, the callee's TM contacts the TMs on those clients directly. Once all needed diffs have been obtained, the callee applies them, in the order in which they were originally generated, to the version of the segment it has cached.

If the TMT is modified by the callee to reflect locks acquired or diffs created during an RPC, the modifications are sent back to the caller along with the RPC results, and incorporated into the caller's copy of the TMT. As in the original call, the code that does this work (`xdr_trans_result()`) is automatically included in the marshaling routines generated by the InterWeave IDL compiler. When the caller needs diffs created by the callee to update its cache, it knows where to get them by inspecting the TMT. Since there is only one active process in a transaction, the TMT is guaranteed to be up-to-date at the site where it is in active use.

Transaction Commits and Aborts

During a commit operation, the library on the client that originally starts the transaction (the transaction *coordinator*) finds all InterWeave clients that participated in the transaction by inspecting the TMT. It then initiates a two-phase commit protocol among those clients by sending every client a *prepare* message. During the first, prepare phase of the protocol, each client sends its locally created and temporarily buffered diffs to the appropriate InterWeave servers, and asks them to prepare to commit. A client responds positively to the coordinator only if all servers the client contacted respond positively. During the prepare phase, each InterWeave server temporarily stores the received diffs in memory.

Once the coordinator has heard positively from every client, it begins the second, commit phase of the protocol by sending every client a *commit* message. In response to this message each client instructs the servers that it contacted during the prepare phase to commit. Upon receiving the commit message, the server writes all diffs to a *diff log* in persistent storage (all non-transaction-based lock releases are treated as commits and the corresponding diffs logged as well), and then applies the diffs to the segments' master copy in the order in which they were originally generated. The persistent diff log allows the server to reconstruct the segment's master copy in case of server failure. Occasionally, the server checkpoints a complete copy of the segment to persistent storage and frees space for the diff log.

A transaction abort call, `IW_rollback_work()`, can be issued either by the application explicitly or by the library implicitly if anything goes wrong during the transaction. Both the InterWeave clients and servers use timeout to decide when to abort an unresponsive transaction. For the sake of simplicity, InterWeave does not provide any mechanism for deadlock prevention or detection. Transactions experiencing deadlock are treated as unresponsive and are aborted by timeout automatically.

Proactive Diff Propagation

Normally, a diff generated inside a transaction is stored on the InterWeave client that created the diff, and is transmitted between clients on demand. To avoid an extra exchange of messages in common cases, however, InterWeave sometimes may send diffs among clients proactively.

Specifically, the TM of an RPC caller records the diffs that are created by the caller and requested by the callee during the RPC session. If the diffs for a segment are requested three times in a row by the same remote procedure, the library associates the segment with this particular remote procedure. In later invocations of the same remote procedure, the diffs for the associated segments will be sent proactively to the callee, along with the TMT and RPC arguments. These diffs are stored in the callee's *proactive diff buffer*. When a diff is needed on the callee, it always searches the *proactive diff buffer* first before sending a request to the InterWeave server or the client that created the diff. When the RPC call finishes, along with the RPC results, the callee returns information indicating whether the proactive diffs are actually used by the callee. If not, the association between the segment and the remote procedure is broken and later invocations will not send diffs proactively. The same process also applies to the diffs created by the callee. If those diffs are always requested by the caller after the RPC call returns, the callee will piggyback those diffs to the caller along with the RPC results in later invocations.

Always deferring propagating diffs to the InterWeave servers to the end of a transaction may incur significant delay in transaction commit. As an optimization, each InterWeave client's TM thread also acts as a "diff cleaner", sending diffs in the *created-diff buffer* to corresponding InterWeave servers when the client is idle (e.g., waiting for RPC results). These diffs are buffered on the server until the transaction commits or aborts.

4. Evaluation

In this section, we use microbenchmarks and a large-scale datamining application to evaluate InterWeave. More results are available in the TR version of this paper [23].

4.1. Transaction Cost Breakdown

We first use a microbenchmark to quantify InterWeave's transaction cost in both local area network (LAN) and wide area network (WAN) environments. In this benchmark, two processes share a segment containing an integer array of variable size and cooperatively update the segment inside a transaction. One process (the RPC caller) starts a transaction and contacts the InterWeave server to acquire a writer lock on the segment (the "lock" phase); increments every integer in the array (the "local" phase); generates a diff that describes the changes it made (the "collect" phase); releases the writer lock (this may be done automatically through options in the RPC call, and does not entail notifying the server); makes an RPC call to the other process, proactively sending the diff along with the RPC call, and waits for the callee to finish (the "RPC" phase).

During this "RPC" phase, the callee acquires a writer lock on the segment (it will find the lock in the TMT, avoiding contacting the InterWeave server); uses the diff in the *proactive diff cache* to update its local copy; increments every integer in the array; generates a diff that describes the new changes it made; and proactively sends the diff back to the caller along with the RPC results.

After the callee finishes, the caller reacquires any locks that might have been released, uses the returned proactive diff to update its local copy (the "apply" phase), prints out some results, and then runs the two-phase commit protocol to update the InterWeave server's master copy of the segment (the "commit" phase). During the "commit" phase, the caller and callee send the diffs they created to the InterWeave server.

We compare the above "proactive transaction" with two other alternatives—"nonproactive transaction" and "no

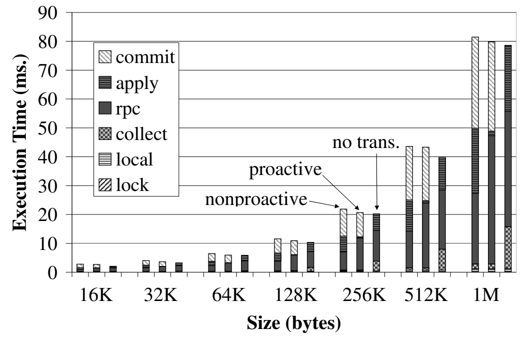

Figure 1. Execution time for transactions that transmit a large amount of data on a LAN.

transaction". With "nonproactive transaction", the diffs are only sent between the caller and callee on demand. During the "RPC" phase, the callee will contact the caller to fetch the diff created by the caller. Likewise, in the "apply" phase, the caller will contact the callee to fetch the diff created by the callee.

With "no transaction" (the basic InterWeave implementation without support for transactions), in the "collect" phase, the caller sends the diff it created to the InterWeave server and releases the writer lock. In the "RPC" phase, the callee contacts the InterWeave server to acquire a writer lock and request the diff it needs. When the callee finishes, it sends the diff it created to the InterWeave server and releases the writer lock. In the "apply" phase, the caller acquires a reader lock and fetches the diff created by the callee from the InterWeave server to update the caller's local copy.

Local Area Network Environment

The first set of experiments were run on a 1Gbps Ethernet. The InterWeave server, RPC caller, and RPC callee run on three different 2GHz Pentium IV machines under Linux 2.4.9. For each configuration, we run the benchmark 20 times and report the median in Figure 1. The X axis is the size (in bytes) of the segment shared by the caller and callee. The Y axis is the time to complete the transaction.

Compared to a "proactive transaction", the "apply" phase in a "nonproactive transaction" is significantly longer because it includes the time to fetch the diff from the callee. Likewise, the "collect" phase and "apply" phase in "no transaction" are longer than those in "proactive transaction", because the diffs are sent to or fetched from the InterWeave server during those phases. For a "proactive transaction", the diffs are sent between the caller and callee during the "RPC" phase. However, the "commit" phase compensates for the savings, resulting in an overall small overhead to support transactions for RPC calls that transmit a large amount of data (see the "proactive" and "no trans." bars).

With the aid of the TMT, processes avoid propagating the diffs through the server when sharing segments. As a result,

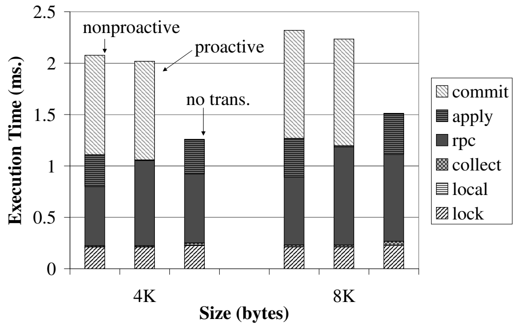

Figure 2. Execution time for transactions that transmit a small amount of data on a LAN.

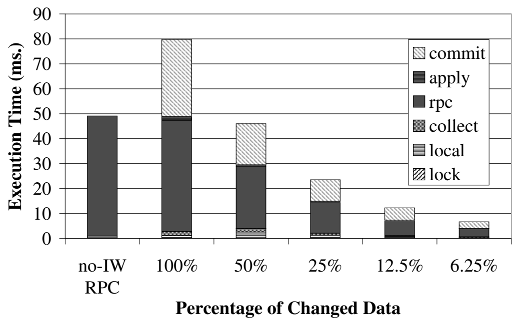

Figure 3. Execution time for "proactive transaction" running on a LAN, when both the caller and callee only update x% of a 1MB segment.

the critical path of the RPC call for a "proactive transaction" (the "RPC" phase) is up to 60% shorter than that of "no transaction" (the "collect"+"RPC"+"apply" phases). In the benchmark, the local computation cost is trivial. For transactions with relatively long computation, "proactive transaction" will send diffs to the InterWeave server in the background, reducing the time spent in the "commit" phase.

The results for smaller segments are shown in Figure 2. The "proactive transaction" has slightly better performance than the "nonproactive transaction" because it saves the extra two round trip times to fetch the diffs. For transactions that only transmit a small amount of data between the caller and callee, the relative overhead of executing the two-phase commit protocol becomes more significant, as seen by comparing with the "no trans." results.

Figure 3 shows the execution time of a "proactive transaction" when both the caller and callee only update x% of a 1MB segment. As the percentage of the changed data goes down, the transaction cost decreases proportionally, due to InterWeave's ability to automatically identify modifications and only transmit the diffs. In all cases, the overhead to compute the diffs (the "collect" phase) is negligible compared with the benefits.

The "no-IW RPC" is a simple RPC program with no use

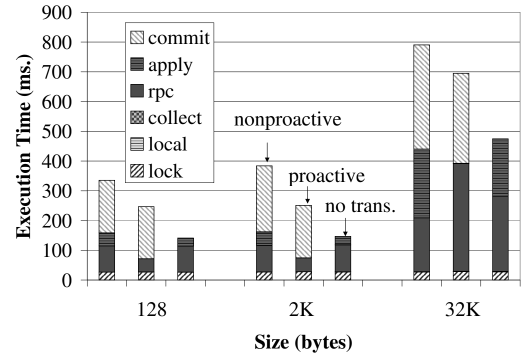

Figure 4. Execution time for transactions running on a WAN.

of InterWeave, sending the 1MB data between the caller and callee directly. It avoids the cost of sending the modifications to an InterWeave server and the overhead of acquiring locks and executing the two-phase commit protocol. The important lesson this figure reveals is that, for temporary (non-persistent) data with simple sharing patterns, it is more efficient to transmit them (using deep copy) directly across different sites than to put them in the global shared space. However, for persistent data (some data outlive a single run of the application and hence must be persistent) with nontrivial sharing patterns, applications can significantly benefit from InterWeave's caching capability. With InterWeave, both deep copy arguments and MIPs to the shared store can be used in a single RPC call, giving the programmer maximum flexibility to choose the most efficient way to communicate data.

Wide Area Network Environment

Our second set of experiments runs the same benchmark on a wide area network. The machines running the InterWeave server, RPC caller, and RPC callee are distributed at the University of Waterloo (900MHz Pentium III, Linux 2.4.18), the Rochester Institute of Technology (300MHz AMD K6, Linux 2.2.16), and the University of Virginia (700MHz AMD Athlon, Linux 2.4.18), respectively.

The execution times of the transactions are shown in Figure 4. They are more than 100× slower than those in the fast LAN. As the data size increases, the relative cost of the "RPC" phase among "nonproactive transaction", "proactive transaction", and "no transaction" changes. When the data size is small, the "RPC" phase in "proactive transaction" is the smallest because it avoids the extra round trip time to acquire the lock or to fetch the diffs. As the data size increases, the diff propagation time, which is included in the "RPC" phase for "proactive transaction", dominates. As a result, the "RPC" phase for "proactive transaction" becomes the longest among the three. Due to the slow network, "no transaction" performs noticeably better than "proactive transaction" as it does not have the overhead of executing the two-phase commit protocol.

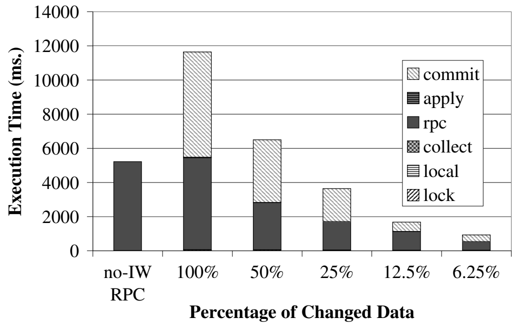

Figure 5. Execution time for a "proactive transaction" running on a WAN, when both the caller and callee update only x% of a 1MB segment.

Figure 5 uses the same settings as Figure 3 except that the experiment is run on a WAN. The results are similar but there are two important differences. First, the savings due to cache reuse are much more significant on a WAN because of the low network bandwidth and long latency. Second, InterWeave's protocol overhead (e.g., diff collection) becomes even more negligible compared with the long data communication time, justifying the use of complex techniques (e.g., diffing and relaxed coherence models) in middleware to save bandwidth for WAN applications.

4.2. Service Offloading in Datamining

In this experiment, we implement a sequence mining service running on a Linux cluster to evaluate the potential performance benefit of combining InterWeave and RPC to build network services, and also to obtain a sense of the effort that a programmer must expend to use InterWeave.

The service provided by the cluster is to answer sequence mining queries on a database of *transactions* (e.g., retail purchases). Each transaction in the database (not to be confused with transactions *on* the database) comprises a set of *items*, such as goods that were purchased together. Transactions are ordered with respect to each other in time. A query from a remote user usually asks for a sequence of items that are most commonly purchased by customers over time.

The database is updated incrementally by distributed sources. When updates to the database exceed a given threshold, a data mining server running in the background uses an incremental algorithm to search for new meaningful sequences and summarizes the results in a lattice data structure. Each node in the lattice represents a sequence that has been found with a frequency above a specified threshold. Given a sequence mining query, the results can be found efficiently by traversing this summary structure instead of reading the much larger database.

We assign one node in the cluster as a front end to receive queries from clients. The front end can either answer mining queries by itself, or offload some queries to other computing nodes in the same cluster when the query load is high. We compare three different offloading strategies. In the first strategy, the front end uses RPC to offload queries to other computing nodes. Each RPC takes the root of the summary structure and the query as arguments. The offloading nodes do not cache the summary structure. This is the simplest implementation one can get with the least amount of programming effort. However, it is obviously inefficient in that, on every RPC call, the XDR marshaling routine for the arguments will deep copy the entire summary structure.

The second offloading strategy tries to improve performance with an ad-hoc caching scheme. With more programming effort, the offloading nodes manually cache the summary structures across RPC calls to avoid unnecessary communication when the summary structure has not changed since the last call. The data mining server updates the offloading nodes only when a new version of the summary structure has been produced. When the summary structure does change, in theory it would be possible for the programmer to manually identify the changes and only communicate those changes in the same way as InterWeave uses diffs. We consider the effort required for this optimization prohibitive, however, because the summary is a pointer-rich data structure and updates to the summary can happen at any place in the lattice. Therefore, this further optimization is not implemented; when the lattice has changed it is retransmitted in its entirety.

Alternatively, the system designer can use the global store provided by InterWeave to automate caching in RPC-based offloading. In this third strategy, we use an InterWeave segment to share the summary structure among the cluster nodes. The data mining server uses transactions to update the segment. When making an offloading call, the data mining server passes the MIP of the root of the summary structure to the offloading node, within a transaction that ensures the proper handling of errors. On the offloading node, the MIP is converted back to a local pointer to the root of the cached copy of the summary structure using `IW_mip_to_ptr`.

Our sample database is generated by tools from IBM research [21]. It includes 100,000 customers and 1000 different items, with an average of 1.25 transactions per customer and a total of 5000 item sequence patterns of average length 4. The total database size is 20MB. The experiments start with a summary data structure generated using half this database. Each time the database grows an additional 1% of the total database size, the datamining server updates the summary structure with newly discovered sequences. The queries we use ask for the first K most supported sequences found in the database ($K = 100$ in our experiments).

We use a cluster of 16 nodes connected with a Gigabit Ethernet. Each node has two 1.2GHZ Pentium III processors with 2GB memory, and runs Linux 2.4.2. One node is

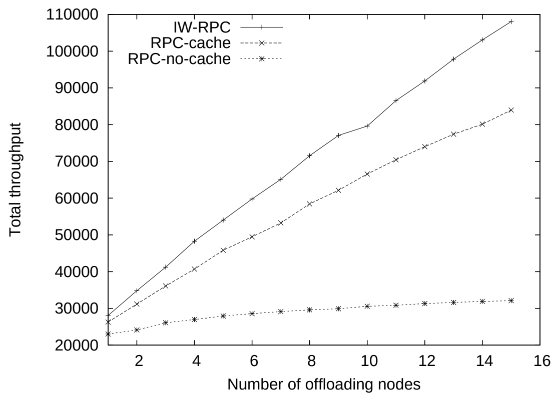

Figure 6. The impact of different offloading strategies on the system throughput.

assigned as the front-end server and offloads incoming user queries to some or all of the other 15 nodes.

Figure 6 shows the aggregate service throughput (i.e., queries processed in 100 seconds) of the cluster with an increasing number of offloading nodes. For each offloading node, the front end runs a dedicated thread to dispatch queries to it. The X axis is the number of offloading nodes we use in the cluster, starting from 0 (no offloading) to a maximum of 15. Beginning with a database of 50% of its full contents, we increase the database to its full size in 50 steps. Each step takes about 2 seconds. The Y axis shows the total number of completed queries during the entire database update process, i.e., 100 seconds. "IW-RPC" uses InterWeave to automate caching of the summary structure. For both the "IW-RPC" and the "RPC-cache" cases, as the database grows, sometimes the cache needs no update while sometimes it does need an update. The reported throughput represents an average over all queries for each case. "RPC-no-cache" uses straightforward RPC offloading with no cache. The throughput for "IW-RPC" scales linearly with the size of the cluster, outperforming "RPC-cache" by up to 28%. Without caching, the system cannot benefit much from using RPC for offloading. Please see the technical report [23] for more detailed results.

5. Related Work

InterWeave finds context in transactional client-server caching protocols [8], traditional databases [9], distributed object caching systems [6, 14], S-DSM systems [1, 2, 26], and a wealth of other work—far too much to document fully here. The following paragraphs concentrate on what appear to be the most relevant systems in the literature. The most prominent features that distinguish InterWeave from previous work are its support for a shared memory programming model across heterogeneous platforms, its exploitation of relaxed coherence models, and its efficient integration of shared state, transactions, and remote invocation.

PerDiS [7] is perhaps the closest to InterWeave among existing systems. It also uses URLs for object naming, supports transactions, and has sharing units equivalent to InterWeave's segments and blocks. PerDiS, however, has no built-in support for heterogeneous platforms, relaxed coherence models, or pointer swizzling. It does not allow remote procedure calls to be protected as part of a transaction.

Smart RPC [13] is an extension to conventional RPC that allows parameter passing using call-by-reference rather than deep copying call-by-value. It uses S-DSM techniques to fetch the referenced data when they are actually accessed. The biggest difference with respect to InterWeave is that Smart RPC does not have a persistent shared store and lacks a well-defined cache coherence model. Because it does not track modifications made by distributed processes, Smart RPC always propagates the parameters modified in the middle of an RPC chain back to the initial caller before making a new RPC. This may significantly slow RPC's critical path. Transactions are not supported in Smart RPC.

Zhou and Goscinski [27] present a detailed realization of an RPC transaction model [9] that combines replication and transaction management. In this model, database clients call a set of remote procedures provided by a database replica to process data managed locally by the replica. InterWeave supports transactional RPC between arbitrary clients and maintains coherence efficiently among dynamically created caches.

Dozens of object-based systems attempt to provide a uniform programming model for distributed applications. Many are language specific (e.g., Argus [15], Mneme [16], and Arjuna [18]); many of the more recent ones are based on Java. Language-independent distributed object systems include Legion [10], Globe [24], Microsoft's DCOM [19], and various CORBA-compliant systems [17]. Globe replicates objects for availability and fault tolerance. A few CORBA systems (e.g. Fresco [14] and CASCADE [6]) cache objects for locality of reference. Unfortunately, object-oriented update propagation, typically supported either by invalidating and resending on access or by RMI-style mechanisms, tends to be inefficient (re-sending a large object or a log of operations). Equally significant from our point of view, there are important applications (e.g., compute-intensive parallel applications) that do not employ an object-oriented programming style.

6. Conclusions

We have described the design and implementation of a middleware system, InterWeave, that integrates shared state, remote invocation, and transactions to form a distributed computing environment. InterWeave works seamlessly with RPC systems, providing them with a global, persistent store that can be accessed using ordinary reads and writes. To protect against various system failures or race conditions, a sequence of remote invocations and data accesses to shared state can be protected in an ACID transaction. Our novel use of the *transaction metadata table* allows processes to cooperate inside a transaction to safely share data invisible to other processes and to exchange data modifications they made without the overhead of going through the InterWeave server.

Experience with InterWeave demonstrates that the integration of the familiar RPC, transactional, and shared memory programming models facilitates the rapid development of maintainable distributed applications that are robust against system failures. Experiments on a cluster-based datamining service demonstrate that InterWeave can improve service scalability with its optimized "two-way diff" mechanism and its global address space for passing pointer-rich shared data structures. In our experiments, an offloading scheme with InterWeave outperforms an RPC offloading scheme with a manually maintained cache by 28% in overall system throughput.

References

[1] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. on Computers*, 37(8):930–945, Aug. 1988.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 152–164, Pacific Grove, CA, Oct. 1991.

[3] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. of the 12th ACM Symp. on Operating Systems Principles*, pages 147–158, Litchfield Park, AZ, Dec. 1989.

[4] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Multi-level Shared State for Distributed Systems. In *Proc. of the 2002 Intl. Conf. on Parallel Processing*, pages 131–140, Vancouver, BC, Canada, Aug. 2002.

[5] D. Chen, C. Tang, B. Sanders, S. Dwarkadas, and M. L. Scott. Exploiting High-level Coherence Information to Optimize Distributed Shared State. In *Proc. of the 9th ACM Symp. on Principles and Practice of Parallel Programming*, San Diego, CA, June 2003.

[6] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *Proc., Middleware 2000*, pages 1–23, New York, NY, Apr. 2000.

[7] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Roberts, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak. PerDiS: Design, Implementaiton, and Use of a PERsistent DIstributed Store. Research Report 3525, INRIA, Rocquencourt, France, Oct. 1998.

[8] M. J. Franklin, M. J. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Trans. on Database Systems*, 22(3):315–363, Sept. 1997.

[9] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1993.

[10] A. S. Grimshaw and W. A. Wulf. Legion — A View from 50,000 Feet. In *Proc. of the 5th Intl. Symp. on High Performance Distributed Computing*, pages 89–99, Aug. 1996.

[11] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Trans. on Programming Languages and Systems*, 4(4):527–551, Oct. 1982.

[12] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(1):109–133, Feb. 1988.

[13] K. Kono, K. Kato, and T. Masuda. Smart Remote Procedure Calls: Transparent Treatment of Remote Pointers. In *Proc. of the 14th Intl. Conf. on Distributed Computing Systems*, pages 142–151, Poznan, Poland, June 1994.

[14] R. Kordale, M. Ahamad, and M. Devarakonda. Object Caching in a CORBA Compliant System. *Computing Systems*, 9(4):377–404, Fall 1996.

[15] B. Liskov. Distributed Programming in Argus. *Comm. of the ACM*, 31(3):300–312, Mar. 1988.

[16] J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Trans. on Information Systems*, 8(2):103–139, 1990.

[17] Object Management Group, Inc. The Common Object Request Broker: Architecture and Specification, Revision 2.0. Framingham, MA, July 1996.

[18] G. D. Parrington, S. K. Srivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. *Computing Systems*, 8(2):255–308, 1995.

[19] D. Rogerson. *Inside COM*. Microsoft Press, Redmond, Washington, Jan. 1997.

[20] M. Scott, D. Chen, S. Dwarkadas, and C. Tang. Distributed Shared State. In *9th Intl. Workshop on Future Trends of Distributed Computing Systems*, San Juan, Puerto Rico, 2003.

[21] R. Srikant and R. Agrawal. Mining Sequential Patterns. IBM Research Report RJ9910, IBM Almaden Research Center, Oct. 1994. Expanded version of paper presented at the *Intl. Conf. on Data Engineering*, Taipei, Taiwan, Mar. 1995.

[22] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Efficient Distributed Shared State for Heterogeneous Machine Architectures. In *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, Providence, RI, May 2003.

[23] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Integrating Remote Invocation and Distributed Shared State. TR 830, Computer Science Dept., Univ. of Rochester, Jan. 2004.

[24] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, Jan.-Mar. 1999.

[25] Xerox Corporation. Courier: The Remote Procedure Call Protocol. Technical Report XSIS 038112, Dec. 1981.

[26] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):540–554, Sept. 1992.

[27] W. Zhou and A. M. Goscinski. Managing Replicated Remote Procedure Call Transactions. *The Computer Journal*, 42(7):592–608, 1999.