BRISK: A Portable and Flexible Distributed Instrumentation System*

Aleksandar M. Bakić, Matt W. Mutka
Michigan State University
Department of Computer
Science and Engineering
3115 Engineering Building
East Lansing, Michigan 48824
{bakicale,mutka}@cse.msu.edu

Diane T. Rover
Michigan State University
Department of Electrical
and Computer Engineering
2120 Engineering Building
East Lansing, Michigan 48824
rover@egr.msu.edu

Abstract

Researchers and practitioners in the area of parallel and distributed computing have been lacking a portable, flexible and robust distributed instrumentation system. We present the Baseline Reduced Instrumentation System Kernel (BRISK) that we have developed as a part of a real-time system instrumentation and performance visualization project. The design is based on a simple distributed instrumentation system model for flexibility and extensibility. The basic implementation poses minimalistic system requirements and achieves high performance. We show evaluations of BRISK using two distinct configurations: one emphasizes isolated simple performance metrics; and the other, BRISK's operation on distributed applications, its built-in clock synchronization and dynamic on-line sorting algorithms.

1 Introduction and Related Work

Designers and users of parallel and distributed systems have applied a variety of monitoring methods and instrumentation techniques to gather information for testing, debugging, and analyzing performance and optimizing systems [11]. These methods and techniques require development of systems for instrumentation data collection, management and analysis, which are themselves distributed systems. A distributed instrumentation system (IS) is typically specialized to an application domain and/or computing environment. Moreover, the distributed nature often makes it harder to use and adapt. A significant investment in time and effort may be needed to understand the IS implementation sufficiently to port and/or configure it for another application and/or environment.

An off-the-shelf distributed IS that is robust, portable and flexible would benefit both designers and users of a wide range of parallel and distributed systems. It would allow them to instrument and begin monitoring their system rapidly, and as needed subsequently, to optimize or extend the IS for their environment. High performance and openness of the IS implementation are equally important for its success as a general-purpose systems software.

Many distributed ISs have been developed over the past decade (e.g., [3, 8, 5]), usually as components of larger software toolkits for analysis of parallel/distributed systems. Only a few have been ported to multiple platforms and made available to broader usage [13, 9, 7, 4, 6]. Most of these contain subsystems that whose services could be provided by a common distributed IS kernel. In this paper we present a basic, reference implementation of a portable and flexible IS called BRISK (Baseline Reduced Instrumentation System Kernel) that we have developed as a part of a real-time system instrumentation and performance visualization project. In the interests of flexibility, BRISK's design is based on a generic distributed IS model that defines three components of a distributed IS: (1) local instrumentation server (LIS), (2) IS manager (ISM), and (3) transfer protocol (TP). For performance reasons, we have based the BRISK LIS implementation on JEWEL's [6] internal and generic external sensors.

Section 2 describes the objectives of BRISK and approaches taken in its design and implementation. Section 3 describes the architecture and implementation details of BRISK. Along with the description, we discuss approaches that provided the IS performance gains. Results of evaluating performance and scalability are given in Section 4.

*This work was supported in part by DARPA contract No. DABT 63–95–C–0072, NSF grant No. CDA–9529488, and NSF grant No. ASC–9624149.

2 Objectives and Approaches

A number of important issues challenge the concept of a portable and flexible distributed IS. These include

- **Degree of intrusion on the application.** Due to the instrumentation overhead, a target system may exhibit different behavior. The overhead should be predictable and must not change the order and timing of critical events in the target system. It is desired that IS components are schedulable with the target system, so that perturbation analyses can be performed to investigate the degree of intrusion.

- **Global clock reference.** Processes that make up a parallel/distributed system run on processors that may have non-synchronized clocks. It is very difficult to determine the precise global time and/or relative timings of events, which is mandatory for determining accurate global states and debugging erroneous timing behavior based on the instrumentation data.

- **Throughput and latency of the instrumentation data transfer.** Events of interest in a target system that are to be processed on-line may together form large volumes of instrumentation data and monopolize IS resources even beyond the target system. On the other hand, in time-critical applications of the IS, it may be desired that important events be delivered to a central place as soon as possible. Clearly, these two requirements are in contradiction.

- **Support for transparent monitoring.** Adding significant amounts of instrumentation code to parallel/distributed systems by users is subject to errors. It is important that tools can be built based on the IS to instrument the target system automatically, so that the users can only specify *what* to monitor, from which aspect, and at which level. The IS design should be flexible enough to allow the development of various such tools.

Our objective was to design BRISK such that it can offer high performance under various requirements, i.e., to make it *flexible* in the performance sense. We chose first to identify simple metrics, related to localized IS performance, and implement, evaluate and optimize relevant parts of BRISK in isolation. More complex IS performance metrics, related to IS operation on distributed applications, depend on the simple ones, plus the target application and system characteristics. Henceforth, we added "tuning knobs" to many of BRISK's subsystems, so that users can trade-off among the various simple and complex IS performance metrics in a specific working environment.

Another aspect of IS flexibility is the ability to support various monitoring methods and techniques through gradual extensions and/or specializations of the basic implementation. BRISK should be able to emulate other methods/techniques (e.g., a hybrid monitoring approach for tracing or profiling) by a software, event-based monitoring approach. On the other hand, the IS should be flexible towards a variety of extant, independently-built tools and systems for the analysis of instrumentation data. Finally, different parallel/distributed applications may require very different instrumentation/experiment scenarios, and the IS should be able to support them. All these issues resulted in the design of BRISK as a general-purpose distributed IS *kernel*, based on a simple model described in Section 1.

The *portability* requirement implies that a distributed IS should be designed and implemented assuming a minimal set of resources that are available in a majority of extant environments. The basic implementation of BRISK relies on a small number of highly available systems libraries, such as those for interprocess communication, XDR and TCP/IP protocols. Users who can afford, for example, synchronized hardware clocks may specialize BRISK for their type of platform, without affecting its portability.

We also believe that the basic IS implementation should be compact, with a comprehensible source code. BRISK consists of two executables and a tiny library.

3 Description of BRISK

3.1 Architecture

On each node of the target system, multiple user processes are instrumented using *internal sensors*. The internal sensors use `cpp` macros to write instrumentation data records to the shared memory. The shared memory is read by an *external sensor*, which runs as another process on the same node and may be assigned a lower priority. Both the internal sensors and the external sensor (EXS) form an LIS that sends instrumentation data to the ISM. Time-stamps, embedded into the instrumentation data records by the internal and external sensors on different nodes, are synchronized. BRISK synchronizes LIS clocks using a modification of Cristian's centralized clock synchronization algorithm [2]. This algorithm invokes a master-slave strategy whereby a master polls the slaves, determines differences between its clock and the slaves' clocks, and updates the slave clocks.

The instrumentation data transfer protocol used between a LIS and the ISM is based on XDR, which makes BRISK amenable to a heterogeneous environment. In the ISM, the instrumentation data records are dynamically sorted online by time-stamp before delivery to consumers. Special, causally-related events are additionally ordered, possibly

overriding incorrect time-stamps. The default output mode of the ISM is writing to a shared memory, which is then read by instrumentation data consumer tools. Besides writing to shared memory, the BRISK ISM may log instrumentation data to trace files in the PICL [12] ASCII format, or it may pass instrumentation data to a list of CORBA-enabled *visual objects* [1]. Figure 1 shows some details of the implementation described below.

3.2 Local instrumentation server (LIS)

The BRISK NOTICE macros are internal sensors used in an application for event notifications. They are an extension of JEWEL `cpp` macros, which write a data record consisting of integers to a ring-buffer data structure in shared memory. The NOTICE macros are capable of writing heterogeneous records, with over ten basic types available for individual fields, ranging from bytes, to floats, to null-terminated strings.

Besides the data types for event data, three system types are available for coordination among BRISK, instrumented distributed applications, and instrumentation data analysis tools. A "time-stamp type," `X_TS`, allows the user to embed BRISK's internal time-stamp into an event record. The embedded time-stamp is an eight-byte `longlong_t`, representing the number of microseconds of Universal Coordinated Time (UTC). The raw local time is obtained by a call to `gettimeofday` library function within the NOTICE macro, which is added to a *correction value* maintained by the EXS, before sending the record to the ISM. The other two system types, `X_REASON` and `X_CONSEQ`, are used to mark causally-related events. The user supplies `u_long` identifiers for fields of these types, determining which *consequence* events must follow respective *reason* events. If BRISK's clock synchronization algorithm fails to prevent the occurrence of tachyons (i.e., a consequence event that appears to happen before its reason event), events marked using these system types will be post-processed by the ISM to ensure proper ordering.

Our goal is to provide the convenience of dynamic typing to new users, and, at the same time, retain the form of a macro for lower intrusion. The header file contains NOTICE macros for up to eight dynamically-typed fields, which we believe are sufficient for many uses. More than eight fields in a macro adds excessive code to a compiled instrumented application, which indirectly increases the intrusion. A utility tool is provided to create custom NOTICE macros having user-defined field types and insert them into the header file. This tool effectively supports an *on-demand* partial evaluation/specialization of NOTICE macros that results in smaller and faster code. It exemplifies the flexibility of BRISK.

3.3 Distributed clock synchronization algorithm

In the Cristian's algorithm, a master polls slaves periodically, in so-called *rounds*. In each round, it queries each slave for its current time, and waits for the answer; when the answer is returned, the master computes the time difference between the pair. This is repeated a number of times for each slave to average the results. At the end of each round, the master sends the time differences to the slaves to adjust their clocks.

The main difference between our algorithm and Cristian's algorithm is that the master (ISM) time is used only as a common reference point for computing relative skews of the slave (EXS) clocks. This is because, for measurement purposes, it is important that the EXS clocks be as close to each other as possible, while it is not necessary for them to be close to the ISM clock. At the cost of small positive drifts of the EXS clocks, the algorithm described below converges faster than the original Christian's algorithm.

Firstly, an EXS clock with the maximum positive skew relative to the ISM clock (i.e., one with the most-ahead clock) is selected, based on polling as in Cristian's algorithm. Then, the skews of the other EXS clocks and their average are computed *relative* to the selected EXS clock (as absolute values), as well as their average. Finally, only the EXS clocks whose relative skews are above the average are advanced by a correction value. This is to account for the network noise and, in a conservative manner, take care not to promote another EXS clock as the fastest one erroneously. The price of this decision is potentially slower convergence. The correction value is chosen as follows: if the average value is above a small threshold, the correction value is equal to the relative skew of the corresponding EXS clock; otherwise, it is a fixed portion of the relative skew (0.7 in the current implementation). This reduction of the correction value is also conservative in nature, because the EXS clocks cannot be perfectly synchronized in practice.

3.4 Transfer protocol (TP)

The XDR protocol has been chosen for the basic BRISK implementation. BRISK's data transfer protocol does not, however, use XDR in the typical way, with `rpcgen` and static typing, as in JEWEL. Instead, each dynamically typed instrumentation data record is sent with a meta-information header needed for it to be correctly received. The external sensor packages instrumentation data in XDR format with the meta-information header compressed, and sends them to the ISM over a TCP stream socket. Minimizing the slack in instrumentation data messages is important since transferring of (likely large volumes of) event records through the network is several orders of magnitude slower than through shared memory.

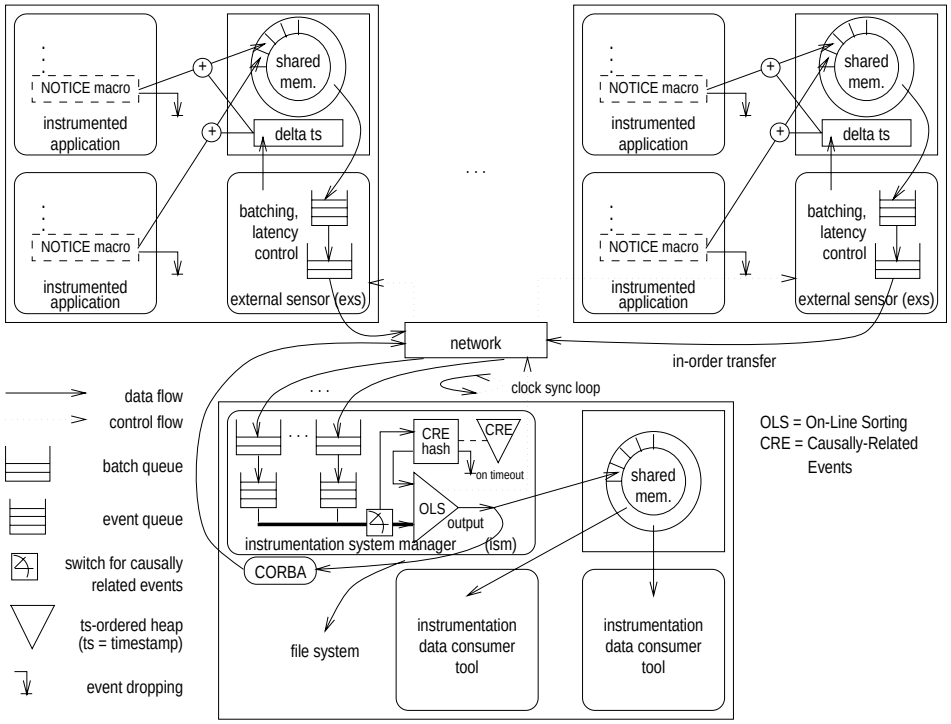

Figure 1. The BRISK basic implementation

3.5 Instrumentation system manager (ISM)

When the ISM receives a data batch from an external sensor, it stores it in the corresponding queue; the in-order arrival of these batches is guaranteed by the socket stream protocol. For dynamic merging/on-line sorting and extracting instrumentation data records from multiple queues, the ISM uses a heap having one entry for each queue.

Each instrumentation data record, after being extracted from the ISM's heap, is written to a shared memory buffer using the same binary structure used by the NOTICE macros. Optionally, a PICL trace record can be generated with the time-stamps either in the UTC format or as the (floating-point) number of seconds since the ISM was run. The visual objects mentioned in Section 3.1 are components of an object-oriented framework for the development of on-line performance visualization of parallel/distributed systems. Through an optionally linked, portable implementation of CORBA 2.0 called MICO [10], the ISM can call remote visual objects' methods and pass instrumentation data records to be processed as PICL strings. Other consumers can read the ISM's shared memory buffer, e.g., using supplied code that creates PICL strings.

3.6 On-line sorting algorithm

Using the synchronized embedded time-stamps, its current time, and a user-specified time frame T, the ISM delays each instrumentation data record for T time units after its creation. If the ISM detects that two successive records from different external sensors have been extracted out of order, it increases the time frame; then, it exponentially decreases the time frame to reduce the amount of instrumentation data delayed in memory. This method of sorting results in a tradeoff between the event ordering and latency.

Additionally, causally-related events are matched via a hash-table: if a consequence event record being processed does not match a reason event record with the same identifier in the hash-table, it is kept in memory until the corresponding reason event record is processed. When a just-arrived reason event record matches a waiting consequence event record whose time-stamp is smaller than that of the reason event, the latter's time-stamp is overriden by a larger value. Since in this case it is obvious that the clocks have not been synchronized (i.e., the time-stamps must reflect the causality), an extra round of the clock synchronization algorithm is invoked immediately. A causally-marked event of either type is kept in memory no longer than a specified timeout, because its peer may have been dropped.

The on-line sorting algorithm assumes that the EXS clocks are perfectly synchronized. Since this is not achievable in practice, tachyon occurrences are possible if the EXS clocks are more apart than it takes for causality information to be passed among nodes. Tachyons can occur if causally-related events in the target application are not marked using BRISK. On the other hand, instrument-

ing some causally-related events using BRISK may help BRISK to keep the EXS clocks better synchronized. This would, in turn, reduce the probability of tachyon occurences related to the other causally-related events, through the extra synchronization rounds.

4 Evaluation of BRISK

We have conducted experiments with BRISK using two configurations. The first configuration consists of one external sensor, and we measured the range of several simple performance metrics. The second configuration includes multiple external sensors on different nodes, and we measured how well the system scales, the quality of the clock synchronization and dynamic on-line sorting. In both configurations, we use simple looping applications instrumented using NOTICE macros having six fields of type `integer`. Including the time-stamp and type information, each instrumentation data record requires 40 bytes in the XDR-based transfer protocol. The experiments were executed primarily on Sun Ultra-1 workstations running Solaris 2.5.1, within a 155 Mbps local ATM network. Measurements using multiple nodes were generally limited to eight nodes, due to the number of simultaneously free workstations available for experimentation. Due to the lack of space, we briefly summarize the measurements.

Simple metrics. The CPU time taken by an average NOTICE macro varied from 3.6 to 18.6 microseconds on three different platforms. The CPU utilization of the EXS on a Sun workstation—where it shares the CPU with the target system—was shown negligible (under 1%) at event rates of up to 38,000 per second. On the other hand, the maximum throughput achieved between an EXS and ISM was 90,000 events per second. Extensive latency measurements (in combination with on-line sorting) are part of future work, but the worst-case lower bound was found to depend on waiting `select` system calls, which can delay an event record for up to 40 ms.

Distributed operation. The CPU demand by the ISM was the bottleneck for achieving high event throughput, but the ISM was able to maintain the maximum aggregate event throughput almost constant with up to 8 EXS nodes. The clock synchronization algorithm was able to keep EXS clocks (8 of them, using 5 s polling period over 10 minutes) within 20–40 microseonds under light working conditions, and most of the time under 200 microseconds at times when disturbances of various sources in the LAN interfered with it. The on-line sorting algorithm was evaluated using streams of artificially delayed event records, and by varying four quantitative and qualitative parameters. We found that setting the time frame T to be as large as the latest late event's lateness is a good strategy for latency-critical applications, and that in all other applications a small exponent constant for reducing T (i.e., a large T's half-life) helps.

Acknowledgments and Further Information

We thank Paul Reed for the work on evaluating BRISK, including many analyses and suggestions that helped us to develop a more robust and efficient implementation of BRISK. The first public version of BRISK is available at `http://www.egr.msu.edu/Pgrt/External/-BRISK-1.0.tar.gz`. It includes a technical report with details of the evaluation and plans for future extensions.

References

[1] A. Bakić, M. W. Mutka, and D. T. Rover. Real-time system performance visualization and analysis using distributed visual objects. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 2 1997.

[2] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[3] P. S. Dodd and C. V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software—Practice and Experience*, 22(10):863–877, October 1992.

[4] W. Gu, G. Eisenhauer, and K. Schwan. Falcon: On-line monitoring and steering of large-scale parallel programs. *IEEE Concurrency: Practice and Experience*, 1998.

[5] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[6] F. Lange, R. Kroeger, and M. Gergeleit. JEWEL: Design and implementation of a distributed measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657–671, November 1992.

[7] B. P. Miller et al. The Paradyn parallel performance measurement tools. *IEEE Computer*, November 1995.

[8] B. P. Miller, C. Macrander, and S. Sechrest. A distributed programs monitor for Berkeley UNIX. *Software—Practice and Experience*, 16(2):183–200, February 1986.

[9] D. A. Reed et al. An overview of the Pablo performance analysis environment. Technical report, Department of Computer Science, University of Illinois, November 7 1992.

[10] K. Römer and A. Puder. MICO: CORBA 2.0 implementation. Technical report, Computer Science Department, University of Frankfurt, Germany, 1997.

[11] J. J. P. Tsai and S. J. H. Yang, editors. *Monitoring and Debugging of Distributed Real-Time Systems*. IEEE Computer Society Press, 1995.

[12] P. H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831-6367, September 1992.

[13] J. Yan. Performance tuning with AIMS—an automated instrumentation and monitoring system for multicomputers. *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, January 1994.