The Omni Macroprogramming Environment for Sensor Networks

Asad Awan, Ahmed Sameh, and Ananth Grama

Department of Computer Sciences, Purdue University, W. Lafayette, IN 47907

Abstract. Structural sensing and control is an important application of the DDDAS paradigm. Our work on structural sensing and control has several key aspects, including model reduction, control, simulation, and validation. Motivated by our work on validation using an actual three-storeyed structure, we are developing a comprehensive systems environment, Omni, for macroprogramming sensor networks. While there have been efforts targeted at enabling programmers to write lean applications for individual sensor nodes, there have been few efforts targeted towards allowing programmers to program entire networks as distributed ensembles. Omni provides an intuitive and efficient programming interface, along with operating system services for mapping these abstractions into the underlying network. In this paper, we provide a high-level overview of the Omni architecture, its salient features, and implementation details. The Omni architecture is designed to be a flexible, extensible, scalable, and portable system, upon which a wide variety of DDDAS applications can be built.

1 Introduction and Motivation

An important application of the Dynamic Data-Driven Application Systems (DDDAS) paradigm is in the control of large civil structures. It is estimated that the United States has an investment of over $20 trillion in its civil infrastructure. While these systems are in constant cycle of deterioration and renewal, they are expected to withstand extreme loads caused by natural disasters as well as human factors. Protecting this investment is of prime social and economic importance. The serviceability and safety of these systems, ranging from high-rise structures and long-span bridges to major pipelines that traverse the U.S., can be greatly improved if damage can be detected and controlled intelligently before catastrophic failures. The technology for detection and control of damage is, in principle, now available through low-power sensors, actuators, and communication and computing elements. As part of our NSF-funded project (ITR/DDDAS), we are developing the necessary computational infrastructure to enable: (a) the effective design and economical construction of highly robust smart structures capable of significantly greater resilience to catastrophic events; (b) enhancing robustness of existing structures by suitably retrofitting smart sensor-actuator complexes; (c) predicting and mitigating the impact of catastrophic events in real time; and (d) off-line data archival and analysis technologies for establishing failure causalities and design enhancements.

We identify several intellectual challenges in the eventual application of closed loop control to structures. We specifically target algorithms and techniques for model reduction and control, analysis and simulation, and systems infrastructure for real-time sensing and actuation. As part of our prior work, we have developed extensive algorithmic and software infrastructure for model reduction and control [16, 10, 8, 6, 9, 2, 3, 5, 4, 14], simulation, and visualization [11, 12, 13]. Our work on simulation and visualizaiton resulted in the first comprehensive science-based analysis of the Pentagon crash. Subsequent efforts have modeled heat-induced structural failure in high-rise buildings. Our current efforts in this direction are aimed at a comprehensive modeling of the structural failure of the World Trade Center. Our modeling and simulation work is complemented by experimental tests at the Bowen Lab of Structural Engineering at Purdue. This unique facility provides us with a powerful validation mechanism for our computational approaches.

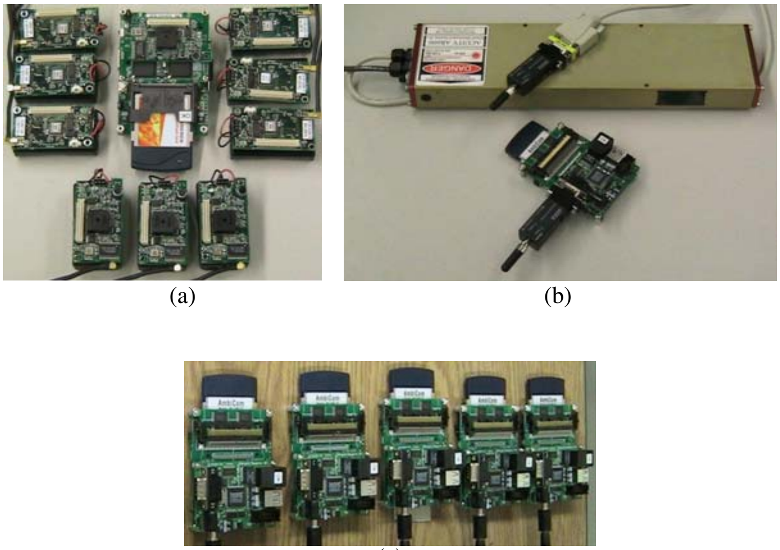
(a) (b) (c)

Fig. 1. Some elements of our current sensing infrastructure: (a) Mica 2 motes with 8MHz Atmega128 μc, ADXL202 Micro-electro-mechanical (MEMs) 2-D accelerometer (range $+/-2g$, sensitivity $12.5\%/g$, $2mg$ resolution at 60Hz, power $< 0.6mA$ at $\sim 3V$); (b) laser displacement sensor, Acuity AR600, accuracy up to 0.04 in, sampling rate up to 1.25KHz; and (c) communication and in-network processing hubs with 400 MHz XScale processors, and communication interfaces for FM, Bluetooth, and 802.11b

As part of these validation efforts, we have built a structural sensing infrastructure based on accelerometers, high-accuracy laser displacement sensors, strain gages, and compute and communication elements (Figure 1). In building this sensing infrastructure, we realized the critical need for a systems environment for (macro)programming sensor networks. This macroprogramming environment is designed for real-time sensing as well as affecting control. It provides support for essential characteristics, including robustness, reconfigurability, and real-time guarantees. In the rest of this paper, we describe the architecture, programming model, and current status of our development efforts towards the Omni environment.

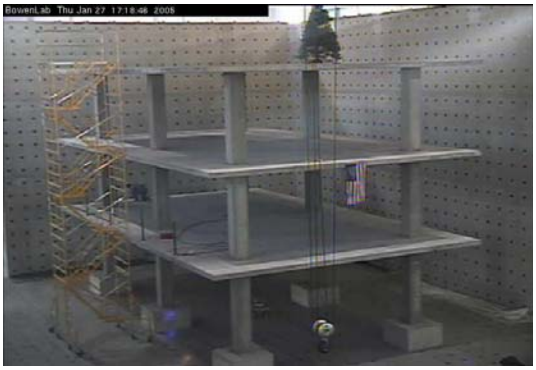

Fig. 2. An instrumented test infrastructure at the Bowen Labs (three story $30' \times 50'$ structure) with extensive actuation and response testing capability

2 Overview of the Omni Macroprogramming Environment

An operating system architecture defines the design paradigm, and hence behavior and performance, of the applications built on top of it. We present a second generation sensor network operating system suite, Omni, which facilitates rapid development of efficient self-organized distributed applications for sensor networks. Existing sensor network operating systems (for example [15, 7, 1]) allow the development of network-enabled applications at each node, however, their design does not directly facilitate the role of application components in the *distributed* behavior of the sensor network as a whole. While these systems address the problem of developing lean (low CPU and memory overhead) applications, they are apathetic to the complexity of designing and implementing high performance distributed functionality in sensor networks. This is a critical requirement for the development of large-scale sensing and control applications.

Other approaches for designing and implementing distributed behavior over sensor networks include the use of domain specific programming languages (for example [20, 19]). However, the use of high level languages often abstracts away low-level, system dependent details, which often need to be tuned, given the resource constraints of sensor nodes. Finally, stream processing database systems for sensor networks ([18, 17]), allow SQL-type queries and aggregation operations over data streams. While this approach affords ease and flexibility, the performance on (typically) resource constrained nodes is limited due to the generality of the database stream management system (DSMS), and customization is limited by the complexity of the design and implementation of the system.

Apart from aiding the design and development of distributed applications, Omni provides several other key features. Similar to SOS [7], we support dynamic update of application components and OS services. However, the distributed design paradigm supported by our system directly allows network wide self-organized adaptation rather than the updation of each module treated independently. Our application development paradigm supports concurrency safety between application components, executing on the same node, irrespective of the underlying scheduling policy or the number of on-board processing elements. The stream processing paradigm allows a programmer to view processing in each component of the application as an independent transaction and relieves the programmer from the burden of maintaining synchronization in a distributed system. Similarly, the stream processing abstraction is ideally suited to the ad-hoc peer-to-peer model of sensor network systems in contrast with client-server based abstractions such as RPC.

3 Omni Architecture

The design of the Omni OS focuses on providing abstractions that allow rapid development of robust and efficient distributed applications for sensor network systems. Our design separates the core OS kernel, OS services, and distributed applications. The core kernel and the OS services control the behavior of a single node, while applications implement the distributed system behavior of the sensor network. Following the stream processing model the applications are developed using a box-and-arc model. Each box represents a *processing element* and arcs represent data channels. However, unlike most systems that are based on this model, we allow runtime reconfiguration of the model, including changing interconnections and replacement of processing elements (PEs). The applications view the sensor network as a single large system, albeit distributed, allowing simplified design. Each processing element is conceptually a single execution unit interfacing only via the input and output channels. A PE can not request dynamic memory except for the dynamically growing input and output queues. In practice, the isolation of a PE provides strong concurrency and memory safety properties. The channels provide transparent and efficient transportation of data streams between PEs in the distributed space (i.e., communicating PEs can be located on the same, or different physical nodes). Channels are implemented as low-overhead lock-free single producer, multiple consumer (SPMC) queues mitigating communication overheads.

3.1 Omni Processing Elements

The design specification of a PE only declares its *functional* behavior and *typed* input and output streams. Therefore, at design time, a PE's relationship with other PEs, or its placement in an application (fundamentally, which is a given instance of the box-and-arc diagram) is not known. The separation of the design of a PE from application design enforces modularization and allows code reuse. Once a repository of functionally specific PEs is developed, a new application can be developed rapidly by designing a connection diagram and providing complementary specifications. These diagrams can be statically checked, e.g., for typed-correctness, to verify that the application meets the distributed behavioral specification. Static checking also removes the burden of verifying interfaces at runtime on resource constrained nodes. In existing sensor network OS examples [7], limited verification is performed at runtime, which affects the safety properties of the system. The expected complexity of interactions between application components further burdens the programmer and makes debugging difficult.

Conceptually a PE is treated as a single unit of execution. At runtime this abstraction is enforced by treating each execution of a PE as a transaction. Each transaction reads input streams, generates output streams, and on success requests the system to commit. A commit involves removing the consumed data from the input streams and making the output streams available for downstream processing. Inability of the OS to perform the commit results in no change to the input or the outputs. This is notified to the PE and a re-execution of the entire transaction follows. This model also allows the system to preempt or *kill* a currently executing PE without generating any inconsistency.

The processing elements are implemented using C language and are compiled using platform specific gcc compiler and linker scripts (to enable dynamic runtime loading). This allows seamless portability of the PEs requiring only recompilation for different platforms. The application is a specification expressed as a schema and is processed by our custom compiler, which in general is platform agnostic, but may be modified to account for optimizations or irreconcilable differences between target platforms. In the future, application design may be generated by a WYSIWYG (what you see, is what you get) GUI front end that generates the schema.

3.2 Omni OS Services

The OS services are designed using the same box-and-arc model, thus are similar to services in a micro-kernel based design. Here, the key difference is that a single multiplexed queue is replaced by point-to-point channels. The connection between services is determined via the box-and-arc configuration rather than use of addresses (which, in practice, are often hard-coded) as in most micro-kernel architectures. In contrast to an application PE, OS services are designed to control the underlying hardware and behavior of each node independently. The operational similarity between PEs and OS services allows a uniform OS runtime architecture, while the design time specifications

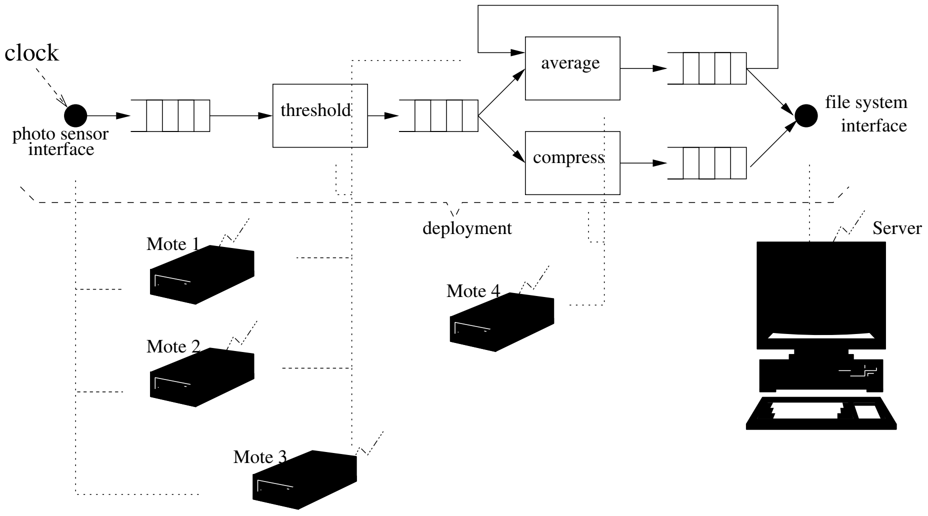

Fig. 3. An example stream processing application connection graph and its deployment over a sensor network system

```
/* ------ A basic application schema ----- */
@  PHOTO_SENSOR_NODE:
TRIGER[Clock, RATE]          -> ADC[PHOTO_SENSOR]

@  NODE_ANY:
ADC[PHOTO_SENSOR]            -> pe_threshold(s_photo_t in)
pe_threshold(s_photo_t out) -> pe_average(s_photo_t in_p)
pe_average(s_avg_t out)      -> pe_average(s_photo_t in_avg+)

@  NODE_FAST_CPUS:
pe_threshold(s_photo_t out) -> pe_compress(s_photo_t in_p)

@ NODE_SERVERS
pe_compress(s_comp_t out_c) -> FILE[COMPRESS]
pe_average(s_avg_t out)      -> FILE[AVG]

/* ------ Threshold function  ------ */
int pe_threshold(flag_t *in_flag, flag_t *out_flag,
               /*in*/ s_photo_t *in, /*out*/ s_photo_t *out)
{
    stream_t *ins, *outs;

    ins  = get_data_stream(in);
    outs = get_data_stream(out);

    while(has_data(ins))
        if (get_data_val(ins) > THRESH)
            set_data(outs, ins);

    commit(out_flag, in );
    commit(out_flag, out);
    return OK;
}
```

Fig. 4. Code sample illustrating the application schema (for the example in Figure 3) and the C Language implementation of the threshold PE

and compile time enforcement maintain the conceptual differences. Note that at the application level, OS services and the core OS are invisible in the application connection diagram. For example, an OS service implementing the routing service is abstracted away by the distributed channel model used by the application PEs. However, the developer still has control over the low-level system components, for example in this case the network implementation of the channel, due to the ability to modify the OS services to monitor and manage system hardware. At the platform level, the core OS kernel provides a hardware abstraction layer (HAL) decoupling the OS services from the platform hardware. Thus, the system design and development is cleanly partitioned into application, OS services, and core OS components.

The OS services and applications (including individual PEs) can be reconfigured and updated at runtime to adapt the performance and functionality of a single node or the entire sensor network system, respectively. The OS also has the capability to autonomically reconfigure the components based on design time instructions and specifications. Providence for terminating an executing PE or OS service at runtime allows updates without sacrificing consistency of the data stream (due to the transaction abstraction).

An example application is illustrated in Figure 3. A connection diagram with different processing elements is shown at the top. At the bottom the deployment of this conceptual connection diagram over the network is illustrated. Different processing elements are assigned to different nodes based on an application schema, shown in

Figure 4). The deployed processing elements are transparently connected by channels in the distributed space. Figure 4 also provides an example implementation of the `threshold` PE in C language.

4 Status of Development Efforts

We are currently developing and exhaustively testing the Omni environment targeted for the AVR (e.g., Mica2 mote) and POSIX platforms. We are also developing utilities for static checking, schema compilation, and over-the-network PE/OS service updation tools. We will port the HAL to other sensor node platforms, allowing reuse of the PE, OS service and most core OS components.

The Omni architecture provides a flexible, portable, and scalable platform over which a number of DDDAS applications can be built. These include data acquisition and sensing, control, as well as analyses applications. We aim to release the entire Omni environment after comprehensive validation, along with exemplar applications (from structural sensing), over the public domain.

References

1. Mate: Programming Sensor Networks with Application Specific Virtual Machines. `http://www.cs.berkeley.edu/~pal/mate-web/`.
2. P.A. Absil, R. Sepulchre, P. Van Dooren, and R. Mahony. Cubically convergent iterations for invariant subspace computation. *SIAM J. Matrix Anal. Appl.*, 2003.
3. Y. Chahlaoui and P. Van Dooren. Benchmark examples for model reduction of linear time invariant dynamical systems. *Model Reduction of Dynamical Systems, Eds. P. Benner et al.*, 2004.
4. Y. Chahlaoui and P. Van Dooren. Model reduction of time-varying systems. *Model Reduction of Dynamical Systems, Eds. P. Benner et al.*, 2004.
5. Y. Chahlaoui, K. Gallivan, A. Vandendorpe, and P. Van Dooren. Model reduction of second order systems. *Model Reduction of Dynamical Systems, Eds. P. Benner et al.*, 2004.
6. Y. Chahlaoui, D. Lemonnier, A. Vandendorpe, and P. Van Dooren. Second-order balanced truncation. *Lin. Alg. Appl.*, 2003.
7. Chih-Chieh Han and Ram Kumar Rengaswamy and Roy Shea and Eddie Kohler and Mani Srivastava. SOS: A Dynamic Operating System for Sensor Networks. In *Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys 05)*, 2005.
8. P. Van Dooren. The basics of developing numerical algorithms. *Control Systems Magazine*, 18-27, 2004.
9. K. Gallivan, X. Rao, and P. Van Dooren. Singular riccati equations stabilizing large-scale systems. *Lin. Alg. Appl.*, 2003.
10. Y. Hachez and P. Van Dooren. Elliptic and hyperbolic quadratic eigenvalue problems and associated distance problems. *Lin. Alg. Appl.*, 371:31-44, 2003.
11. C. Hoffmann, S. Kilic, V. Popescu, and M. Sozen. Integrating modeling, visualization and simulation. *IEEE Computating in Science and Engineering*, January/February 2004.
12. C. Hoffmann, S. Meador, S. Kilic, V. Popescu, and M. Sozen. Producing high-quality visualizations of large-scale simulations. *IEEE Visualization*, 2003.

13. C. Hoffmann and V. Popescu. Fidelity in visualizing large-scale simulations. *Computer-Aided Design*, 2006. To appear.
14. B.F. Spencer Jr., S.J. Dyke, M.K. Sain, and J.D. Carlson. Phenomenological model of a magnetorheological damper. *ASCE Journal of Engineering Mechanics*, 2006. To Appear.
15. Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An Operating System for Sensor Networks. *Ambient Intelligence*.
16. W.H. Liao and C.Y. Lai. Harmonic analysis of a magnetorheological damper for vibration control. *Smart Mater. Struct.*, 11:288-296, 2003.
17. Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. In *Proceedings of TODS*, 2005.
18. Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of OSDI'02*, December 2002.
19. Ryan Newton, Arvind, and Matt Welsh. Building up to Macroprogramming: An Intermediate Language for Sensor Networks. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, April 2005.
20. Ryan Newton and Matt Welsh. Region Streams: Functional Macroprogramming for Sensor Networks. In *Proceedings of the First International Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.