Flow Webs: Architecture and Mechanism for Sensor Webs

Michael M. Gorlick Samuel D. Gasster Grace S. Peng Michael McAtee

The Aerospace Corporation
El Segundo, California
Email: gorlick@aero.org, gasster@aero.org
January 23, 2007

***Abstract*— Our vision of sensor webs—dynamic amalgamations of sensor webs each constructed within a flow web infrastructure—embraces sensors webs as "systems of systems" in which many sensor webs of many kinds—geophysical, routing, service, and computational—are interlinked on demand as needs and circumstances dictate. Flow webs, are by philosophy, design, and implementation, a *dynamic* infrastructure that permits massive adaptation in real-time. Flows may be attached to and detached from services at will, even while information is in transit through the flow. This concept, *flow mobility*, permits on-the-fly integration of earth-science products and modeling resources in response to real-time demands. Flows themselves rely upon the periodic streaming exchange of computational state among peers. This exchange of computational state is informed by Computational REpresentational State Transfer (CREST), a generalization of web practices in which web requests for content and content-centric web responses are replaced by the symmetric exchanges of mobile code and computational continuations among peers.**

I. Introduction

The "sensor web," like the internet and the world wide web, will be a *federation* of many sensor webs, large and small, under many distinct spans of control, that loosely cooperate and share information for many purposes with no single infrastructure or designated overarching authority.

"The" sensor web will not likely be established top-down by fiat. Realistically, it will grow piecemeal as distinct, individual systems are developed and deployed, and as legacy systems are retrofitted with "sensor web friendly" front-ends. The development arc of a federated sensor web will mirror that of the world wide web, a pastiche of sites and services, with only some expressly built for a sensor web. Therefore, the architecture of the sensor web is of fundamental import; architectural strictures that inhibit innovation, experimentation, sharing, or scaling may prove fatal.

Sensors are not limited to environmental, radiometric, or meteorological phenomena. Sensor readings include the health and status telemetry of remote in-situ sensors, the RFID-enabled inventory of radiosondes at a local launch site, the available excess processing capacity of forecasting clusters, or even "network weather," the responsivity, capacity, and delay of the sensor web network, in short, sensors measure both the physical environment *and* the virtual environment of the sensor web complex.

Thus, sensor webs are *domain-specific*. For example, *geophysical* sensor webs measure geophysical phenomena for earth scientists; *computational* sensor webs measure server load, memory consumption, disk capacity, backplane bandwidth, and environmental parameters (rack temperatures, air flow rates, cooling reserves) for the providers of large-scale computing infrastructure; and *routing* sensor webs measure packet flow, queuing delay, jitter, bit error rates, and bandwidth consumption, for network providers. A large, integrated geophysical sensor web as envisioned by NASA [15] requires computational and routing sensor webs as well to allocate and manage the computing and network infrastructure necessary for modeling, prediction, data archive, feedback, and sensor control.

Finally, specialized sensor webs will arise, each with a particular focus. For example, within Southern California it is clear, given politics and historical precedent, that an "air quality" sensor web would be managed as an independent entity, perhaps affiliated with, but separate and distinct from, a "wildfire" sensor web or a "weather forecasting" sensor web. Consequently, any realistic sensor web architecture must expressly accommodate a diverse cooperative of sensor webs just as the modern world wide web seamlessly permits innumerable diverse and distinct web sites, each with its own purpose, perspective, and content.

A number of standards based on the technology of XML and web services are under consideration for sensor webs [22]. Seeking to abstract and simplify sensor access these standards apply XML metadata (to describe) and standard HTTP methods or SOAP methods (to access) sensor-centric services. Alternatively, we approach sensor webs from the perspective of *active services* [10], a generalization of web services. Active services are implemented by programs authored by network and service users. Consequently, service users need not rely on service vendors to implement the services they desire. Active service architectures introduce additional flexibility and powers of adaptation well above and beyond that of classic web services including user-specific and user-directed transcoding, translation, modeling, dynamic service composition, distributed discovery, distributed tasking, service transactions and coordination, service caches and proxies, as well as service streaming.

We approach active services in the context of Computational REpresentational State Transfer (CREST), a generalization of the modern web whose focus is the exchange of mobile code and computational continuations among peers [8], [14]. In CREST web URLs denote computational resources, namely Scheme [6] interpreters whose environments contain URL-specific bindings (name/value pairs). Those bindings may be data or functions or both. For example, the Scheme binding environment for the URL

`www.weather-data.us/US/CA/San-Diego/station12/`

may contain the functions `wind-speed` and `wind-direction` whose return values are the most recently measured wind speed (in meters-per-second) and wind direction (in degrees). Other URLs hosted at `www.weather-data.us` or elsewhere offer Scheme interpreters whose top-level binding environments contain entirely different bindings with different names and different values. Scheme, a well-known and rigorously defined Lisp-dialect, has the virtue that the representation of data and programs is identical, that is, data and programs are both represented as lists of cells. Scheme's uniform representation of data and programs, combined with its powerful macro facility, allow Scheme programs to easily generate other Scheme programs—an effective tool in a web whose only coin of exchange is programs.

Web requests in CREST are arbitrary Scheme programs issued by the requestor that are executed by the Scheme interpreter hosted at the URL that is the target of the request. Web responses in CREST are either Scheme programs or Scheme continuations (themselves reified as Scheme programs) to be interpreted in the environment of a Scheme interpreter elsewhere in the web. A continuation $c_{P,t}$ is the representation of the execution state of program P (including its execution stack and control registers) at time t. Executing $c_{P,t}$ at some later time t' resumes the execution of P at exactly the point at which it left off at time t. The programs (or continuations reified as programs) returned as responses may be interpreted (exected) by the requestor or sent themselves as requests to other URLs.

Similarly, a server receiving a request (namely a Scheme program or its continuation) may forward that request to another URL for interpretation, execute the request (program/continuation) and return the outcome (program/continuation) to the requestor, or, in mid-execution, forward the continuation of that execution to another URL for interpretation. Further, the recipient of any program/continuation may meta-interpret the program/continuation (since from the Scheme perspective programs and data are synonymous) and the outcome of that meta-interpretation may be itself another Scheme program. Consequently, both CREST endpoints and intermediaries may inspect, analyze, wrap, or modify the requests and responses (Scheme programs/continuations) they encounter thereby encouraging the promulgation of novel caches and proxies.

The CREST model and its implementation as a universe of URL-dependent Scheme interpreters simplifies and streamlines the CREST equivalent of HTTP (the HyperText Transport Protocol). For example, reading the current wind speed s and direction d from a San Diego meteorological station as a list $(s\ d)$ requires only

```
(GET
   "www.weather-data.us"
   "/US/CA/San-Diego/station12/"
   '(list (wind-speed) (wind-direction)))
```

which means send the Scheme program (expression)

```
(list (wind-speed) (wind-direction))
```

to the URL

`www.weather-data.us/US/CA/San-Diego/station12`

for execution (interpretation) and return the outcome to the requestor.

In the CREST architectural style new web services are constructed by users who dispatch Scheme programs to web URLs. Figure 1 illustrates a simple user-created service extension implemented as a Scheme expression (program) to gather and return five timestamped wind speed and direction readings, taken at fifteen second intervals. It is important to note that Scheme programs and expressions are the assembly language of the CREST-enabled active web. The vast majority of users will employ higher-level tools and libraries, written in a variety of other languages, that generate Scheme programs for exchange and evaluation among web peers. For example, in the experimental sensor webs that we are constructing the bulk of the implementation will be in Python and a Python/Scheme bridge will hide the Scheme-specific details of CREST exchanges from sensor web middleware and applications.

II. Flow Webs and Flows

Using the active services of the CREST architectural style we are developing a system architecture, the *flow web*, that elevates *flows*, sequences of messages over a domain of interest and constrained in both time and space, to a position of primacy as a dynamic, real-time, medium of exchange for computation and services. The flow web captures, in a single, uniform architectural style, the conflicting demands of sensor webs including, dynamic adaptation to changing conditions, ease of experimentation, rapid recovery from the failures of sensors and models, automated command and control, incremental development and deployment, and integration at multiple levels, in many places, at different times.

Flow occurs over a definite, finite span of time and is unidirectional, oriented in "network space" from a given source to a given sink (destination). We treat flows as fundamental network objects; flows may be named, detached from and connected to network access points, filtered, merged, combined, and rate-controlled. Informally, a flow is akin to a network "garden hose" where the flow has two distinguished endpoints, the *source* and *sink* of the flow (the two ends of the "garden hose") and an identity independent of the content moving

```
(begin
   ;; Define a function to take samples at fixed intervals
   ;; and return those samples as a list.
   (define sample
      (lambda (n interval)
         (let ((outcome '()))
            ;; Take n readings.
            (do ((i 1 (+ i 1)) (> i n))
               ;; Assemble a list of timestamped readings.
               (set! outcome
                  (cons
                     (list (wind-speed) (wind-direction) (now))
                     outcome))
               ;; Wait interval seconds between each reading.
               (wait interval))
            outcome)))  ;; Return the list of speed/direction/timestamp samples.
   ;; Take 5 readings at 15 second intervals.
   ;; Return the readings in ascending temporal order.
   (reverse (sample 5 15)))
```

Fig. 1. A Scheme program to gather and return five timestamped readings (taken at fifteen second intervals).

through it. The endpoints may be attached and detached as need and circumstances dictate even while data is in transit; a property we term *flow mobility*.

Flows are the connective tissue of flow webs, massive computational engines organized as directed graphs whose nodes are semi-autonomous components and whose edges are flows. A component may be a source of zero or more flows and/or the sink of zero or more flows. Components consume and generate flow content; for example, a component might consume periodic transmissions of Level 0 sensor telemetry and produce processed Level 1 sensor data for flow transmission to downstream components.

The individual components of a flow web may themselves be encapsulated flow webs, in other words, a flow web subgraph may be presented to a yet larger flow web as a single, seamless component. Flow webs, at all levels, may be edited and modified while still executing. Within a flow web individual components may be added, removed, started, paused, halted, reparameterized, or inspected. The topology of a flow web may be changed at will, since the flows that interconnect the individual components may, with equal alacrity, be added, removed, disconnected, and reconnected. Flow webs, modeled as arbitrary directed graphs, may easily contain feedback cycles, a crucial element of model-driven interaction and adaptation within sensor webs. Flow webs are explicitly intended to be modified on-the-fly, an attribute well-suited for dynamic interactions in sensor webs.

Flow webs offer three distinctive advantages over the request/response behaviors of HTTP and web services:

- *Continuous data transfer* among peers in which each flow sink (component) may adjust the reliability of data transfer in keeping with sink-specific semantics, the rate of data generation, and the bandwidth of the network connection between the flow source (component) and the sink.
- *Selective exchange* in which the flow sink is free to narrow the flow in time, space, or content as the flow proceeds—in short, the sink may select from the flow exactly the data that it requires for its purposes.
- *Arbitrary intermediation* in which intermediate components may be inserted anywhere within flows at any time for transformation and translation, buffering, resampling, or augmentation.

We anticipate that flow webs will: permit rapid, incremental development; encourage experimentation and diversity; offer exceptional visibility into and control over distributed computations; and scale gracefully to support massive sensor and computational networks. Flow webs are also compatible with other specialized service architectures including sensor networks, grid computing, and web services. Thus flow webs may significantly ease the development and deployment of large, integrated sensor webs.

Sensor networks composed of thousands of small, wireless, in-situ devices equipped with an array of environmental probes [23] have attracted tremendous research interest over a broad range of topics including deployment [5], localization [16], time synchronization [12], and wireless communications and protocols [20]. Data aggregation, the combination of individual sensor readings into a cohesive stream, nis now recognized as an important function of sensor networks [17], [18]. Flows address a different set of problems, adaptation, feedback, and reuse, than those faced by individual sensor networks.

Grid computing is a computational model that relies upon many networked computers to offer a virtual architecture that distributes execution across a parallel infrastructure [9]. Problems in grid computing include sharing heterogeneous

resources (operating systems, hardware platforms, and hosts), services discovery, security, resource allocation, and programming languages for parallelization. Flow webs are one of many possible overlays atop a grid infrastructure. In other words, grid infrastructure may provide the computational resources required by large-scale flow webs.

Web services, "a software system designed to support interoperable machine-to-machine interaction over a network," [24] are a form of remote-procedure call (RPC) and rely heavily upon SOAP, the Simple Object Access Protocol, a wire format for remote method calls and responses. Recent studies suggest that SOAP imposes substantial overhead [4] or requires extensive customization for scientific computation [7] and may be inappropriate for applications characterized by large quantities of streaming data such as atmospheric data for numerical weather prediction. The flow web protocols are far less burdensome than those required by web services and it is straightforward to encapsulate a web service as a flow, allowing flows to be run as an overlay or intermediary service.

A. Flow Characteristics

Flows appear repeatedly in networks and applications. Flow contents, timescales, and rates vary widely. The aperiodic flow of an RSS feed may have an average frequency best measured on a time scale of days, but with a high burst bandwidth, while sensor-driven flows may have a periodic frequency of tens of Hertz (for example, 30 frames-per-second video) and a sustained bandwidth of hundreds of kilobits to megabits per second. Flows possess distinct units of content, persist over time, exhibit aperiodic and periodic behavior, may appear and disappear, are not necessarily known in advance, and demonstrate 1-1, 1-n, m-1, and m-n relations among producers and consumers.

Flow framing is a critical element of flow generation and interpretation and reflects many considerations including rate control, the constraints and characteristics of network transport, and content semantics. For example, the framing structure of video codecs such as MPEG-4 or Ogg Theora [19], abstractly cast as a sequence

$$I_0, \delta_{0,1} \ldots, \delta_{0,m}, I_1, \delta_{1,1} \ldots, \delta_{1,m}, I_2, \ldots$$

where full frames I alternate with a series of deltas δ to be applied in order against the immediately preceding full frame, is a flow framing pattern that appears in many domains.

Flows are not necessarily frequent or periodic, for example, event distribution is marked by our inability to know in advance when an event may occur. On the other hand, periodic flow admits of a rich temporal semantics. The calculation of flow integrity and reliability is a complex combination of flow periodicity, transport, and framing structure. For example, lower transport reliability may be tolerated in a high-rate, redundant flow such as streaming video where occasional missing content elements are ignored, interpolated, or reconstructed. Far more stringent requirements may be levied against a flow transporting infrequent but vital events, such as process control alarms or those events affecting human life or safety.

III. Flow Implementation

Flow mobility [11] is a distinguishing characteristic of flow webs and gives flow webs the ability to interconnect disparate services on-the-fly. Its implementation requires *stateless connections* in the style first proposed by Aura and Nikander [1], [2] and recently implemented for TCP by Shieh, Myers, and Sirer [21]. The continuation exchanges fostered by CREST simplify the construction of stateless connections.

We briefly sketch a CREST-based implementation of receiver-side flow mobility. Assume, without loss of generality, that a source S generates a collection of sensor readings at a regular rate $r \leq 100$ Hz. Let each such timestamped collection be a frame f. At a minimum the source retains the most recent frame f_i for at least a period $1/r$. In practice, most sensor sources will retain the m most recent frames $f_i, f_{i-1}, \ldots, f_{i-(m-1)}$. Let R be a flow receiver. R dispatches a Scheme program p (or equivalently a continuation c_p) to the URL denoting S.

The interaction between S and p proceeds as follows:

1) S invokes p (c_p) with a list $(f_i \ f_{i-1} \ \ldots \ f_{i-(m-1)})$ of the available frames
2) p (c_p) returns to S a frame selection $j, i \geq j \geq i-(m-1)$ and a continuation c_p for p. p selects just those frames f_j required by R (since p was dispatched (created) by R it is well-informed with regard to R's requirements)
3) S invokes c_p with the packets $p_{j,0}, p_{j,1}, \ldots, p_{j,n-1}$ that constitute frame f_j.
4) p transmits the packets $p_{j,0}, p_{j,1}, \ldots, p_{j,n-1}$ to receiver R via UDP using a combination of rate-based and congestion-control mechanisms to ensure TCP-friendliness [13]
5) If p receives one or more negative acknowledgements from R then the missing packets $p_{j,k}$ are retransmitted ahead of any other waiting, untransmitted packets
6) p either halts or returns a continuation c_p to S. If the former then flow transmission is complete and S ceases to execute p. If the latter then S loops to step (1) and continues

On the receiver side R estimates connection bandwidth and congestion from the received frame packets $p_{j,k}$ and periodically streams to R's program p executing at S the ongoing connection flow state (all flow state history is maintained at the receiver R). R uses negative acknowledgements to notify p at S of any missing packets that it requires. p executing at S always knows the network location of R from the return address contained in the UDP packets that R periodically transmits to S containing flow state or negative acknowledgements. Note that if receiver R fails to receive a packet $p_{j,k}$ for a frame f_j it is not required to request retransmission from p executing at S as it may be capable of tolerating, or recovering from, the loss without any additional assistance from p. In this manner, R achieves exactly the degree of flow reliability that it requires, taking into account the frame generation rate, the available

bandwidth, and network congestion along the path from S to R.

The receiver R may transport itself to another network location (URL U) by generating a continuation c_R and transmitting c_R to U for execution. c_R, since it is a continuation, contains all of the state required to immediately restablish contact with S and redirect the ongoing flow to the new sink U. In the meantime another continuation c'_R remains behind at R to simultaneously buffer and subsequently forward on, to the new incarnation c_R of R at U, those frame packets $p_{j,k}$ that were enroute when R decamped for U. In other words, c'_R establishes a short-lived "miniflow" from R to c_R at U to minimize the losses. Once the original $S \rightarrow R$ flow is drained and c'_R has all of the $S \rightarrow R$ frame packets that it wants then R halts execution. At this point the sink of the flow from S is anchored at U.

Sender-side flow mobility employs comparable mechanisms as a server may force a server-side flow endpoint to generate a stateful continuation c_R for migration to another network location and leave behind another continuation c'_R to mop up any unfinished transmissions required by the receiver. It is even possible to move both endpoints nearly simultaneously since their temporary doppelgangers, left behind as continuations at the original endpoints, can assist to re-establish the flow between the two new endpoints.

IV. Summary

Like the sensor web enablement of the Open Geospatial Consortium [3] we intend to exploit a rich collection of middleware and services for the construction of large-scale sensor webs. However, we have a distinctly different view of the structure and implementation of the requisite web services. Using the CREST architectural style we suggest implementing web services as the ongoing exchange of Scheme programs and Scheme continuations (reified as Scheme programs) among web peers, yielding an *active* web services network. Those active services are the foundation for an implementation of mobile flows, streams of information whose endpoints (source and sink) may be moved from one network location to another in a manner that is transparent to higher-level middleware and applications. From flows we extrapolate the architecture of flow webs, directed graphs of active components whose edges are mobile flows, that may be edited, modified, and "rewired" during executing while still preserving computational and communication integrity. Sensor webs, realized as flow webs, may exhibit many attractive and useful properties including dynamic adaptation and feedback, redundancy and fault-tolerance, ease of modification and execution transparency.

V. Acknowledgements

This work is supported by NASA Award AIST-QRS-06-0047. The CREST architectural style was formulated by Justin Erenkrantz, University of California, Irvine and Michael M. Gorlick.

References

[1] T. Aura and P. Nikander, "Stateless connections," Research Report A46, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, 27 pages, May 1997.

[2] T. Aura and P. Nikander, "Stateless connections," Proceedings of the First International Conference on Information and Communication Security (November 11 - 14, 1997), Y. Han, T. Okamoto, and S. Qing, Eds. Lecture Notes In Computer Science, vol. 1334. Springer-Verlag, London, 87-97, 1997.

[3] M. Botts, A. Robin, John Davidson and Ingo Simonis, "OpenGIS Sensor Web Enablement Architecture Document," Open Geospatial Consortium, OGC-06021r1, 2006-03-04.

[4] Kenneth Chiu, Madhusudhan Govindaraju and Randall Bramley, "Investigating the Limits of SOAP Performance for Scientific Computing," IEEE International Symposium on High Performance Distributed Computing, 2002.

[5] S. S. Dhillon and K. Chakrabarty, "Sensor Placement for Effective Coverage and Surveillance in Distributed Sensor Networks," IEEE Wireless Communications and Networking Conference, March 2003.

[6] R. Kent Dybvig, "The Scheme Programming Language," Third Edition, The MIT Press, Cambridge, Massachusetts, 2003.

[7] Robert A. van Engelen, "Pushing the SOAP Envelope With Web Services for Scientific Computing," Proceedings of the International Conference on Web Services, 2003.

[8] Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana and Richard N. Taylor, "Harmonizing Architectural Dissonance in REST-based Architectures," Institute for Software Research, Technical Report UCI-ISR-06-18, University of California, Irvine, December 2006.

[9] Ian Foster and Carl Kesselman, "The Grid: Blueprint for a New Computing Infrastructure," Morgan Kaufmann Publishers, November 1998.

[10] Michael Fry and Atanu Ghosh, "Application Level Active Networking," Computer Networks, 31(7), 655–667, 1999.

[11] Michael Gorlick, "Streaming State Kinematics and Flow Engineering," Institute for Software Research, Technical Report UCI-ISR-06-3, University of California, Irvine, March 2006.

[12] J.V. Greunen and J. Rabaey, "Lightweight Time Synchronization for Sensor Networks," International Workshop on Wireless Sensor Networks and Applications, San Diego, California, September 2003.

[13] Yunhong Gu, Xinwei Hong, and Robert L. Grossman, "Experiences in the Design and Implementation of a High Performance Transport Protocol," Supercomputing 2004, Pittsburgh, Pennsylvania, November 2004.

[14] David A. Halls, "Applying Mobile Code to Distributed Systems," PhD Thesis, University of Cambridge, Cambridge, England, June 1997.

[15] G. Higgins et al., "Advanced Weather Prediction Technologies: Two-way Interactive Sensor Web & Modeling System," Phase II Vision Architecture Study, NASA ESTO, November 2003.

[16] X. Ji and H. Zha, "Sensor Positioning in Wireless Ad-hoc Sensor Networks Using Multidimensional Scaling," INFOCOM, March 2004.

[17] Xin Liu, Qingfeng Huang, Ying Zhang, "Combs, Needles, Haystacks: Balancing Push and Pull for Discovery in Large-Scale Sensor Networks," ACM Conference on Embedded Networked Sensor Systems, November 2004.

[18] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," Fifth Symposium on Operating System Design and Implementation, December 2002.

[19] "Theora I Specification," Xiph.org Foundation, May 6, 2005.

[20] Joseph Polastre, Jason Hill and David Culler, "Versatile Low Power Media Access for Wireless Sensor Networks," ACM Conference on Embedded Networked Sensor Systems, November 2004.

[21] Alan Shieh, Andrew C. Myers, and Emin G. Sirer, "Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility," Proceedings of Symposium on Networked Systems Design and Implementation, Boston, Massachusetts,May 2005.

[22] "Sensor Web Enablement," Open Geospatial Consortium (`www.opengeostatial.org`), OGC White Paper, OGC 06-046r2, July 24, 2006.

[23] Malik Tubaishat and Sanjay K. Madria, "Sensor Networks: An Overview," IEEE Potentials, April/May 2003.

[24] "Web Services Glossary," `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/`.