5-2005

Live Data Views: Programming Pervasive Applications that Use "Timely" and "Dynamic" Data

Jay BLACK
IBM T. J. Watson Research Center

Paul CASTRO
IBM T. J. Watson Research Center

Archan MISRA
Singapore Management University, archanm@smu.edu.sg

Jerome WHITE
IBM T. J. Watson Research Center

DOI: https://doi.org/10.1145/1071246.1071294

Live Data Views: Programming Pervasive Applications That Use "Timely" and "Dynamic" Data

Jay Black,[*] Paul Castro, Archan Misra, Jerome White[*1]
IBM T.J. Watson Research Center, Hawthorne, NY
jpblack@uwaterloo.ca, castrop@us.ibm.com, archan@us.ibm.com, jerome@cs.caltech.edu

ABSTRACT
In the absence of generic programming abstractions for dynamic data in most enterprise programming environments, individual applications treat data streams as a special case requiring custom programming. With the growing number of live data sources such as RSS feeds, messaging and presence servers, multimedia streams, and sensor data, a general-purpose client-server programming model is needed to easily incorporate live data into applications. In this paper, we present Live Data Views, a programming abstraction that represents live data as a time-windowed view over a set of data streams. Live Data Views allow applications to create and retrieve stateful abstractions of dynamic data sources in a uniform manner, via the application of intra- and inter- stream operators. We provide details of our model and evaluate a proof-of-concept Live Data Views implementation to monitor traffic conditions on a highway. We also provide the preliminary design of a J2EE-based implementation, and outline some of the research challenges raised by this abstraction in a distributed computing environment.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures—data abstraction, patterns; D.1.3 [**Programming Techniques**]: Concurrent Programming—concurrent programming, distributed programming

General Terms
Algorithms, Design, Measurement

Keywords
J2EE, dynamic data, stream operations, EJB, middleware

1. INTRODUCTION
A class of applications is emerging for effectively monitoring and adapting to the dynamic state of physical or virtual environments. This vision of large-scale monitoring applications has been embraced in several domains of pervasive and mobile computing, such as telematics, context-aware computing, business-process optimization, and environmental tracking (e.g., forest fires, intrusion detection). These applications obtain and harness data from heterogeneous sources that have *liveness* properties. Live data has a notion of *currency*, where recent values subsume older ones. Live data values are also dynamic relative to the lifetime of an application session. Liveness in data implies a notion of data elements as ephemeral entities: when a data element becomes sufficiently stale, it has negligible utility and may be entirely disregarded. For example, an application that warns drivers of traffic congestion needs only the most up-to-date status reports, and may safely discard older status reports. (The obvious value of mining historical data is beyond the scope of this paper).

Currently, applications that process live data use hard-coded logic that is not shared by other applications. Since each live data source requires unique logic for processing, any combination of live data from different sources must be treated as a special case. Moreover, enterprise-grade programming environments, such as J2EE or .NET, lack direct infrastructural elements or programming constructs to support the easy incorporation of live data into applications. A suitable "data-services" middleware component, exposing a portable and application-independent programming model, would allow applications to delegate much of the low-level data-processing logic to an opaque lower layer, and use a common, and hopefully simple, programming abstraction to easily incorporate live data of different types and from multiple sources into applications.

In this paper, we present a programming abstraction for applications using live data. Since conventional databases are not optimized to support the high data-update rates often exhibited by such data sources, live data is viewed through the prism of streams. In this model, data sources frequently "push" typed data onto the network (e.g., formatted in XML) to be used by applications. Recent work, such as Telegraph [3] and Borealis [1], has looked at generic system-level abstractions and performance optimizations needed by data management systems that process data streams. However, a programming abstraction for connecting these systems to actual applications remains an open issue. We bridge the gap between the system-level stream-processing mechanisms and the programming tasks that application developers perform to incorporate live data as a first-class data type. Moreover, we introduce challenges related to scalability and

MDM 2005 May 9–13, Ayia Napa, Cyprus

[1] Authors listed alphabetically

[*] Work performed while authors were visiting IBM Research. Jay Black is currently at the University of Waterloo. Jerome White is currently at the California Institute of Technology.

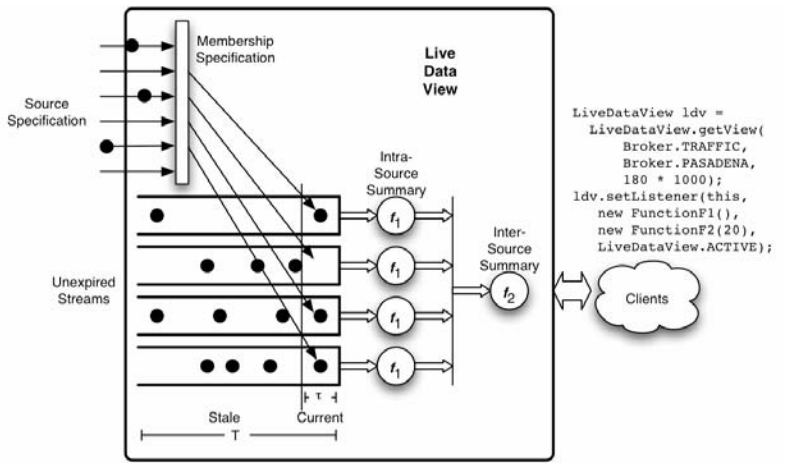

Figure 1. Diagram of a Live Data View and example code. In the call to 'getView', the programmer defines SSpec, MSpec and T. It is in the instantiation of the function object, f_2, that the window duration, τ, is specified.

consistency that a runtime infrastructure for such a programming abstraction must address

Our programming model represents live data as a set of streams over which we define a set of operators. Our fundamental data abstraction is a Live Data View (LDV), which provides a time-window view on a set of streams, each representing a distinct data source. An LDV includes an explicitly timed and sequenced set of data elements from individual streams; the set evolves with the passage of time and the arrival of new stream elements. Live Data Views provide the following:

- *Dynamic state-based "views"* of data streams. Applications that use live data typically monitor the state of a certain set of data sources that satisfy some functional criteria (such as those hospital patients currently posting critical alarms or those instant-messaging users currently in a particular office location). The LDV allows applications to create a dynamic state view, expressed via stream-based semantics. This is similar to proposals for data-replica management using soft-state protocols [10], except that our model generalizes the state representation across multiple data sources.
- *Expression (summarization) of state* through intra- and inter-stream operators. The application state is defined as the result of an operator applied to the data sequence within each individual stream, followed by an operator across streams to summarize the collective state. For example, a stream that reports the current temperature of a hot spring can be summarized via average, max or min operators on temperature values in the last half hour. This can drive a top-k operator that reports the values (or IDs) of the 5 hottest springs.
- *Measures* of divergence between dynamic state replicas. Liveness implies that not only data values, but also the freshness of the data, contribute to replica divergence.

As a proof of concept, we have developed two traffic-monitoring applications that infer the current traffic levels on a highway using processed images from live video feeds. Our sample applications illustrate an important benefit: the LDV provides a generic and reusable server-side component that significantly lowers the communication overhead observed by application clients that use the derived "state," as opposed to custom clients that operate on the raw streaming data. In the remainder of this paper, Section 2 formalizes the definition of the Live Data View, and presents the parameters used to define a specific LDV. Section 3 then describes how our model is incorporated into an application. Next, Section 4 describes and evaluates our webcam-based traffic monitoring applications. Section 5 then presents challenges with implementing our model within an enterprise programming environment, and with building consistency metrics for managing replicas of Live Data Views. We present related work in Section 6. Section 7 concludes the paper with a discussion of future work.

2. OVERVIEW OF THE LDV MODEL

A live data view (LDV) provides access for clients to simple, current state information, derived from a large, changing set of independent message sources. An LDV is designed to impose minimal temporal and reliability constraints on its sources, and to be highly scalable. As shown in Figure 1, the model is based on an abstract two-dimensional matrix, with time along one dimension and sources along the other, a notion of "current" information, and two orthogonal summarization operations providing simple state that clients can obtain by direct queries or subscription to change notifications.

We assume all sources of a particular type provide messages that conform to a common XML schema, and that the LDV can uniquely identify each source. Messages are stamped by the source with a sequence number that increases monotonically; sequence numbers from two different sources are incomparable, and sources do not have synchronized clocks. Due to unreliable message delivery, messages from a single source may be reordered, delayed, or dropped. However, to provide some notion of current data, all messages are also timestamped by the LDV on receipt, that is, on entry at the top left corner of Figure 1. A "membership specification" filters arriving messages based on per-message attributes, resulting potentially in fewer sources and messages.

After timestamping and filtering, each message is added to the stream of recent messages for its source. The most recent message arriving no more than τ seconds ago is considered the current message for the stream, if any. Messages older than T seconds expire and are discarded, as are empty streams. The messages in unexpired streams are used to calculate intra-source summaries

(function f_1), and the client-visible state of the LDV is then calculated by applying the inter-source summary, f_2, to the results.

Formally, an application views its interaction with the LDV in terms of the specification of the following components:

- *Source Specification* (SSpec) indicates the set of sources of interest. This is usually defined as a specification of the schema that each source exposes, and the semantic meaning associated with the source data values (e.g., values from TRAFFIC sensors in the figure).
- *Membership Specification* (MSpec) indicates which messages should be processed by the LDV, and is defined as a predicate on the values of various attributes of the message (e.g., traffic sensors in PASADENA).
- T is the "time to live" of messages in the LDV, calculated from the timestamp on entry, and τ is the time an arriving message remains current unless it is superceded by a more recent message from that stream.
- *Intra-Stream Summary Operator* (f_1) indicates the computation to be performed over the sequence of elements in each unexpired stream. Examples of f_1 include average or exponential average (over all data elements), sum (over all elements) or current (which provides the current element, if any).
- *Inter-Stream Summary Operator* (f_2) indicates the computation to be performed over the results of f_1. f_2 captures the creation of state across streams, and may be used either to reduce the set of relevant stream values (e.g., a top-10 or max operator), or to fuse values of different sources (e.g., derive a probabilistic estimate of "intruder detection" using readings from multiple cameras).
- *Specification operator* (Spec) indicates what LDV state events trigger a corresponding notification to the application. This may include "active" (notification of any state change) or "delta" (notification when the state changes by an appropriate threshold or percentage), defined over the output of f_2.

The LDV model is clearly much simpler, compared to the semantics of conventional messaging systems (e.g., [6]) or stream operators. This is the outcome of a conscious effort to define a "bare-minimum" abstraction that can be implemented within a conventional enterprise programming environment such as J2EE, while possessing enough semantic richness to support a large class of "event-monitoring" live-data applications.

The LDV model has a rather loose definition of "time," without strict temporal or reliability guarantees. This is intentional—our programming model is directed towards applications such as environment monitoring that have no hard real-time constraints. Moreover, in a practical implementation, each source would presumably publish its data elements using best-effort APIs (e.g., the Java Messaging Service (JMS) Publish Subscribe Interface [11]) with little or no coordination with other sources. Imposing reliable delivery semantics (such as guaranteed, once-only delivery) in a distributed environment requires fairly complex messaging-systems infrastructure (e.g., Gryphon [2]), which seemed to be overkill for many of our target applications. The loose and very flexible use of timestamps makes the LDV model useful mostly in general-purpose, "best-effort" enterprise programming environments, and is too lax for applications where fine-grained relative time differentiation is critical. Moreover, as sequence numbers have no cross-stream significance, an LDV cannot be used for precise temporal correlation of streams. The LDV model implicitly assumes that both T and τ are reasonably large compared to the stream arrival rate, so that phase effects between streams are not a real concern. Thus, LDV provides a much more relaxed notion of consistency than stateful messaging-oriented middleware (e.g., SMILE [7]), and intuitively aims to manipulate physical state that is certainly evolutionary, but not singleton-transient. For example, in our sample traffic-monitoring application, the occasional loss of a "congestion level" report from a particular webcam source is not critical; the application is really interested more in the medium-term state, rather than singleton data elements. Moreover, while traffic congestion levels change over minutes, they will certainly not change radically between successive readings from a webcam reporting every 30 seconds, and clock offsets of a few seconds between different camera readings are unimportant.

The explicit use of orthogonal summary functions f_1 and f_2 restricts an LDV to cases where the derived state is decomposable along the two axes. Functions f_1 and f_2 are driven by observations on the emerging category of sensor-driven applications. f_1 can be viewed as a smoothing operator that eliminates the impact of noisy singletons, typically generated by error-prone or unreliable sensors. f_2, on the other hand, can be viewed not just as a means of filtering to reduce the client traffic (e.g., selecting the top two congested road sections), but also as a way to construct derived state from individual sensor values.

3. CLIENT PROGRAMMING WITH LIVE DATA VIEWS

There are two distinct aspects to Live Data Views: the client-side programming model and the server-side infrastructure needed to maintain it. In this section, we describe the programmatic abstraction for using an LDV to incorporate live data into an application. We discuss server-side issues in Section 5.

From the application-client perspective, LDV programming is similar to database programming, where a client establishes a connection to a database, and then specifies a data structure (e.g., a *rowset*) that represents a view over data in the database. Unlike database programming, however, the LDV client must remain in communication with the server to receive updates. We assume this is done through asynchronous messaging to reduce the number of open network connections. To reduce the overhead of this communication, an LDV client is updated only if the real LDV state changes significantly, as expressed by the Spec parameter.

To complete the server-side specification of an LDV, the programmer specifies the SSpec, the MSpec, the window duration T, and τ (see Figure 1). As elements come into the LDV, they are scheduled for removal after T seconds. The SSpec and the MSpec define the type of information the client is interested in, and these are passed on to the server, which is responsible for the initial processing of streamed data. For example, the SSpec could be a *topic name* and the MSpec could be a simple predicate over attribute-value pairs, as in JMS.

The LDV server component collects data to represent the dynamic state; the programmer can then summarize the state by specifying f_1 and f_2. Common statistical operators over numerical values such

as average, max, and min, would be included as part of the standard LDV operator package. Alternatively, for example, a programmer can implement function objects that process XML data from RSS feeds by extending the default f_1 and f_2 classes. Thus, the decision to extend f_1 or f_2 depends solely on the application. All function objects must implement an `apply()` method, which is called by the LDV when a value is added to the array. The `apply()` method in an f_1 object is designed to work over each stream of the entire LDV, while that of the f_2 object is designed to work over the return type of f_1. During the operation of the LDV, dynamic alterations to function objects and update policies are allowed.

The client interacts with an instantiated LDV by either synchronous calls for the state, or via subscriptions for specific events in, or changes to, the LDV's state. Examples of these events include "current value for source s_1 has changed," "current value for f_2 has changed," "current value for f_1 for source s_1 has changed", "data element for s_1 has expired," etc.

4. REAL-TIME HIGHWAY TRAFFIC MONITORING

In this section, we describe our proof-of-concept, Java-based implementation of an LDV system that keeps track of traffic conditions on a highway. We have developed two related applications: one that requires the conditions of the top 20% of the "currently congested" roadways, and another that wishes to be apprised of roadway sections where congestion is building up (i.e., "average" recent delays that have a positive derivative). Monitoring traffic is something that lends itself well to the LDV concept, as road conditions change constantly and traffic information is readily available.

In our prototype, we implemented a software-based traffic "sensor" that periodically downloads the latest JPEG image of a particular highway section from a public, web-based highway camera. The sensor exists off-board from the client and "streams out" status reports to LDVs. This report includes the location of the traffic sensor, the relative level of traffic (high, medium, low), and a co-efficient used internally by the traffic sensor to determine traffic levels. Our sensor calculates traffic congestion levels by employing edge detection and subtraction of successive images as a measure of motion. For our purposes, we use cameras available for highways in Connecticut [4] and Seattle [13], and download images at various rates, though the maximum update rate for a traffic camera is typically once every 30 to 90 seconds.

To find areas containing the most traffic, we used an f_1 which kept a moving average of traffic values for each camera, while f_2 returned the top 20% of those moving averages. To find areas of increasing traffic, f_1 computed the difference between moving average values of the sensor reports for each individual stream, while f_2 extracted all values that were positive. Both clients were "active," meaning they subscribed to automatic updates from the LDV.

To measure the data passing through the system, we monitored the three areas in which data is exchanged: at the filter, the LDV, and the client. The filter was the first point of entry into the LDV for information sent by sensors. It exists within the LDV framework, and its purpose here was to ensure that error values from the sensors did not make their way into the LDV. The filter also deals with out-of-order messages. Measuring the data flow into the filter is significant because it represents the input of raw data from the sensor. Without the presence of the LDV, this data would go directly to the client. With our nominal filtering, the traffic client in both applications received, on average, 50% or less of the data sent from the sensors.

5. LDV SERVER RUNTIME IMPLEMENTATION AND CHALLENGES

A distributed server-side implementation of the LDV abstraction is not a trivial task, and must address three particular challenges.

- *Source Scalability*: The LDV must be capable of scaling to a very large number (O(10,000) and above) of potential data sources.
- *Many Packet Arrival and Timer Events*: The LDV must also be able to deal with the potentially high rate of timer-based events (such as data expiration or staleness) imposed by the time-based model.
- *Time-Based Weak Consistency*: Different clients or application instances may have different tolerances for divergence from the "current" state of the LDV.

We now outline an approach to dealing with the first two challenges through the use of "industry-standard" runtime environments, and return later to the third.

The J2EE programming model [12], including Enterprise Java Beans (EJBs), defines a runtime architecture for component-based enterprise applications. They execute in a server "container" infrastructure that provides consistency, scalability, concurrency, and distribution. The EJB model is presently geared towards "static" data, principally stored in backend databases. However, an appropriate combination of the functionality of various EJBs allows us to develop a runtime component that supports the LDV abstraction over data streams, while still leveraging the underlying scalability of the J2EE container.

Figure 2 presents only one of several possible approaches to optimizing an LDV implementation across multiple clients, where multiple clients of the same type of dynamic data are likely to possess identical notions of liveness (same T and τ), and differ only in the choice of operators. The investigation and evaluation of alternative J2EE-based runtime implementations (e.g., a separate LDV for each client), required for other forms of client heterogeneity, is part of ongoing work.

The Spec threshold in the LDV model allows a particular client's "view" of the state to diverge from the true "time-windowed" state by a tolerance threshold D. This is really a form of weak consistency that can reduce the traffic volume between an LDV server and the client. For example, we measured the message

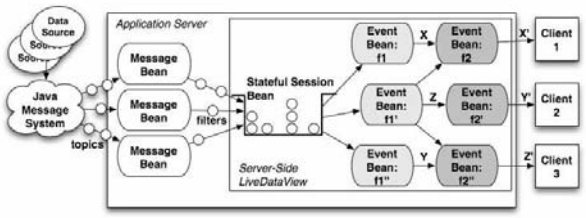

Figure 2. Server-side implementation of a Live Data View using J2EE

volume of our "top 20%" traffic application for different values of D, where D represents the minimum amount of change in consecutive f_2 results that warrants an update of the client. Even a small value of D reduced the traffic volume by 20%, while larger values of D provided savings up to 40%. This helps increase the scalability of LDVs.

Moreover, defining tolerance over an LDV in terms of a divergence measure introduces a new time-based weak-consistency model across multiple clients. The consistency semantics are also crucial for efficiently maintaining multiple replicas of the same LDV on different servers in a distributed runtime environment. We believe that research into this novel concept of "consistency over time-derived views" will create new consistency semantics that complement the extant work [5][13], which supports multiple consistency models over either data from individual streams or over time-independent database relations.

6. RELATED WORK

Applications centered on monitoring the physical environment are prevalent in many domains, including the wireless sensor community. Mainwaring *et al.* [8] presents a reference architecture for habitat monitoring of microclimates, where distributed sensors generate data streams that are processed by backend servers. LDV should provide a useful programming tool for clients that use such sensor information. Network monitoring is another application area that could benefit from LDVs.

Using highly dynamic data requires specialized data-stream processing architectures optimized for rapid updates. Researchers are investigating scalability and modeling issues for stream processing systems. Telegraph [3] and Borealis [1] both provide a fundamental set of relational-like operators that can be applied to streams and both look at adaptive techniques to optimize the throughput of query processing. LDV extends this work to provide a simplified application-level abstraction of dynamic data as a set of streams. One main goal of LDV is to improve programmer productivity when using live data in an application; additionally, its server-side components can be used to reduce communication costs as well as the overhead of a stream processing system through judicious LDV replica management.

LDV not only attempts to model streams, but acts as the representation of dynamic state. In this sense it is closely related to messaging work such as Gryphon [2] and SMILE [7]. In particular, SMILE is an overlay for a messaging network that captures state using a relational model. SMILE is designed to support applications that require a stricter notion of data fidelity (e.g. banking applications) by providing somewhat ACID-like guarantees. LDV differs from SMILE in its fundamental modeling approach: LDV is designed for applications that can tolerate some amount of imprecision as found in [9].

7. CONCLUSIONS AND FUTURE WORK

The LDV model is still in early stages of development. As part of ongoing work, we are investigating the set of "standard" operators that can support a large number of applications. Additionally, we will try and uncover deeper primitives that may functions as operators between LDVs (e.g. a join operator).

Scalability issues remain a major focus for our work. We plan to investigate alternative server-side architectures that can work seamlessly with clients to process live data. Overall, combining a scalable architecture with intelligent replication support remains an open and challenging research problem. In addition, we are also interested in identifying and prototyping other applications that will require our technology.

8. REFERENCES

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack , JH Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[2] M. K. Aguilera, R. Strom, D. Sturman, M. Astley, T. Chandra. Matching events in content-based subscription systems. In *Proc. 18th Annual ACM Symp. on Principles of Distributed Computing (PODC)*, 1999, 53-61.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. 1st Biennial Conf. on Innovative Data Systems Research (CIDR) 2003.*

[4] Connecticut Department of Transportation, http://www.conndot.ct.gov.

[5] S. Cuce and A. Zaslavsky, Supporting multiple consistency models within a mobility enabled file system using a component based framework. *Mobile Networks and Applications 8*, 2003, pp 317-326.

[6] P. Eugster, P. Felber, R. Guerraoui, A. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv. 35*(2): 114-131 (2003).

[7] Y. Jin and R. Strom. Relational subscription middleware for internet-scale publish-subscribe. In *Proc. 2nd International Workshop on Distributed Event-based Systems (DEBS)*, 2003, pp 1-8.

[8] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. 1st ACM Int. Workshop on Wireless Sensor Networks and Applications (WSNA) 2002*, 88-97.

[9] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. 2003 ACM SIGMOD*, 563-74.

[10] S. Raman, S. McCanne: A model, analysis, and protocol framework for soft state-based communication. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 1999, pp. 15-25.

[11] Sun Microsystems. Java Message Service. http://java.sun.com/products/jms/

[12] Sun Microsystems. Java 2 Platform, Enterprise Edition. http://java.sun.com/j2ee

[13] Washington State Department of Transportation, http://www.wsdot.wa.gov

[14] H. Yu and A. Vahdat, Design and evaluation of a continuous consistency model for replicated services, *ACM Trans. on Computer Systems 20*(3), Aug 2002, 239-282.