

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, it is necessary to develop a general-purpose client-server programming model to 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 preliminary design guidelines on a J2EE-based implementation, and mention some of the research challenges raised by this abstraction in a distributed computing environment.

1. Introduction

As the physical world becomes more networked, a class of applications is emerging that centers on effectively monitoring, and adapting to, the dynamic state of physical or virtual (computing) 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 recently generated data values subsume older data values. 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 on the highways needs only the most up-to-date status reports regarding traffic conditions. Clearly, recent reports are more directly valuable than past values of traffic conditions. Older status reports can safely be discarded by the application (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 a) delegate much of the low-level data-processing logic to an opaque lower layer, and b) 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 [1] and Aurora [2] 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.

In this paper, we attempt to bridge the gap between the system-level stream processing mechanisms and the programming tasks application developers perform to incorporate live data as a first-class data type. Our programming model represents live data as a set of

^{*} 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.

¹ Authors listed alphabetically

streams over which we define a useful 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 that 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 [3], 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 a 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.
- *Metrics* for calculating divergence between dynamic state replicas. Consistency mechanisms for replica management traditionally measure the divergence between two replicas based on value. Liveness

dictates that not only data values, but also the freshness of the data contribute to replica divergence.

As a proof of concept, we develop 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 incorporates a two-dimensional framework with time along one dimension and sources along the other, a notion of “current” information, and two orthogonal summarization

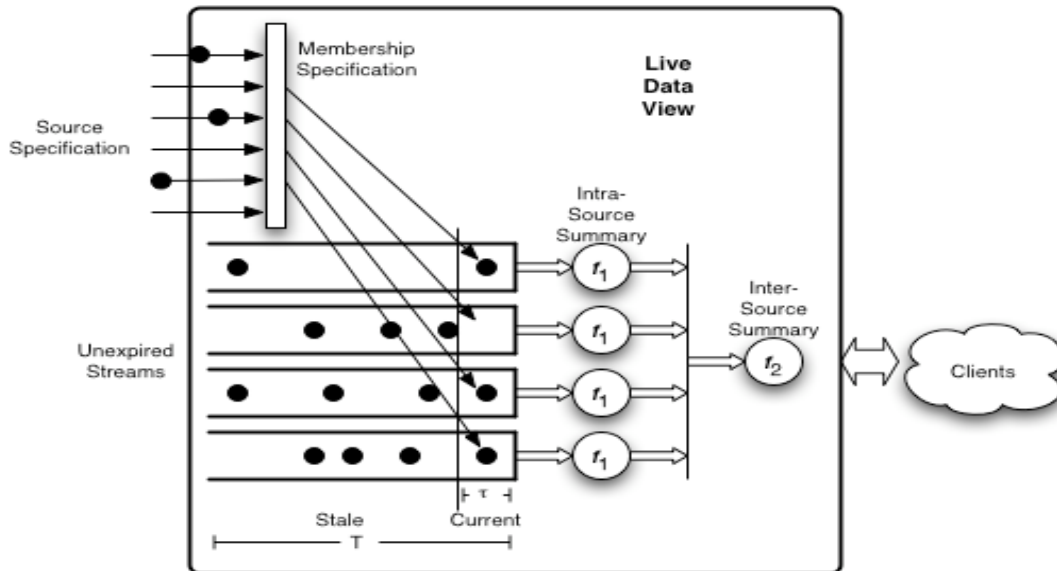


Figure 1. Diagram of a Live Data View

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 “temperature sensors”).
- *Membership Specification* (MSpec) indicates which messages should be processed by the LDV. Typically, this is defined as a predicate on the values of various attributes of the message (e.g., sensor values with “location = New York”).
- 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). Intuitively, this operator performs functions such as smoothing and outlier elimination, because of the noisiness associated with sensor-generated values.
- *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 by utilizing values 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, not shown in Figure 1) indicates what LDV state events trigger a corresponding notification to the application. This may include “all” (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., [4]) 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., JMS Publish-Subscribe) 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 [15]), which seemed to be overkill for many of our target applications. Indeed, 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.

LDVs also allow very flexible use of timestamps. Given our target of instantiating an LDV-based run-time on general-purpose, “best-effort” enterprise programming environments, our model is too lax for applications where fine-grained relative time differentiation is critical. Thus, an LDV would not be suitable for answering questions such as “was stock A more than 20% higher than stock B at time instant X?” However, there is a large body of applications where such fine-grained differentiation is unnecessary. Moreover, 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. As sequence numbers have no cross-stream significance, an LDV cannot be used to answer questions such as “did stream A get its 50th–60th packets within the same interval as stream B’s 50th–60th packets”? Thus, LDV provides a much more relaxed notion of consistency than stateful messaging-oriented middleware (e.g., SMILE [5]), and intuitively aims to manipulate physical state that is certainly evolutionary, but not singleton-transient. For example, while traffic congestion levels change over minutes, they will certainly

```

1  ...
2  LiveDataView ldv =
3  LiveDataView.getView(Broker.TRAFFIC,
4                        Broker.PASADENA,
5                        300 * 1000);
6  ldv.setListener(this,
7                  new FunctionF1(),
8                  new FunctionF2(),
9                  LiveDataView.ACTIVE);
10 ...

```

Figure 2: Client pseudo code to instantiate an LDV. The programmer first gets an instance of the LDV (lines 2 through 5). She then sets the current object as an “active” listener (line 9) on that LDV (line 6), providing f_1 and f_2 objects (lines 7 and 8) to operate on the state.

not change radically between successive readings from a webcam reporting every 30 seconds.

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. As a counter-example, this model cannot be used to express semantics such as “max of the 5th element of stream A and nearest-in-time element of stream B.” In practice, we believe that most applications are indeed decomposable. Moreover, 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—in many instances, individual sensor readings are error-prone and unreliable. 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. In fact, most touted examples of sensor-based applications use aggregate state observed across multiple sensors (e.g., the average reading from all temperature sensors in a region), rather than individual sensor readings (even if they are smoothed).

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 cannot be disconnected once the view is obtained, but must remain in communication with the server. To reduce the overhead of this communication, an LDV can be updated only in the case of a significant change to its state, as we describe in Section 5.

To complete the server-side specification of an LDV, the programmer specifies the SSpec, the MSpec, the window duration, and T (Figure 2, lines 3-5 respectively). 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 the Java Messaging Service (JMS) Interface [6]. An example is SSpec = “highway traffic sensor” and MSpec = “location = Pasadena, CA.”

The LDV server component collects data to represent the dynamic state; the programmer can then summarize the state by specifying the f_1 and f_2 operators as function objects. Programmatically, all other function objects are derived from one of the two base classes for intra- and inter-stream operators. The base classes implement the identity function and simply act as pass-through filters for data elements. A programmer extends these objects to provide further functionality. Common statistical operators over numerical values such as *average*, *max*, and *min*, would be usually included as part of the standard LDV operator package available to all clients. However, functions are not restricted to numerical components. For example, a programmer can implement function objects that process XML data from RSS feeds.

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 the entire LDV, while that of the f_2 object is designed to apply to 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. Figure 2 shows a sample client-side code snippet for a Java-based traffic monitoring application based on the LDV model.

4. Real-time Highway Traffic Monitoring

In this section, we describe our proof-of-concept Java-based implementation of an LDV-compliant “live-data” system that keeps track of traffic conditions on a highway. We have developed two related applications: one in which an application (e.g., a car) seeks to know the conditions on the top 20% of the “currently congested” roadways, and another in which the application wishes to be apprised of roadway sections where congestion is building up (i.e., “average” recent delays have a positive

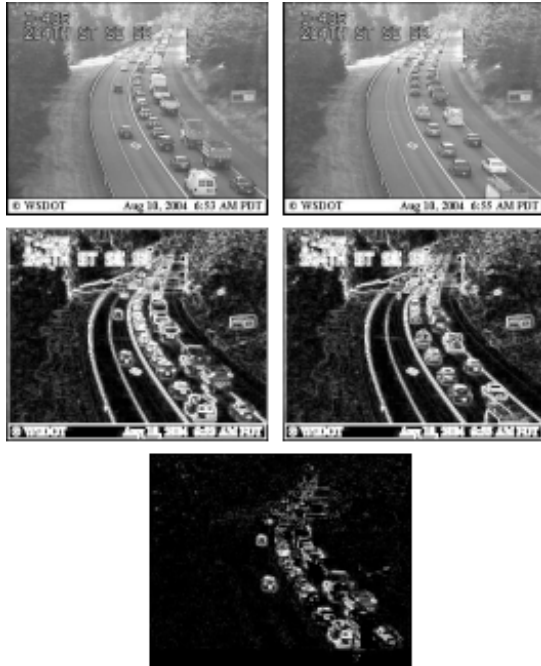


Figure 3. This is a detailed look at our traffic detection method. The top row contains two unaltered consecutive images from our traffic cameras. Beneath these images are their edge detection equivalents. The image at the bottom is the result of the subtraction operation; its mean pixel value is approximately 13.8.

erivative). 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 implementation, a traffic sensor first periodically downloads the latest JPEG image of a particular highway section from a public, web-based highway camera, and then “streams out” a status report to our LDV. 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. For our purposes, we use cameras available for highways in Connecticut [7] and Seattle [8], and download images at various rates, though the maximum update rate for a traffic camera is typically once every 30 to 90 seconds.

To demonstrate our prototype system, we had to implement an image analysis technique that would take a traffic image as input and estimate the associated congestion level. As our intent is more to demonstrate our dynamic data-management middleware than to develop a novel image-analysis algorithm, we used a simple method for determining traffic levels (more robust methods for inferring traffic levels from images have been reported, for example [9]). The image is first put through the Frei and Chen edge detection algorithm [4], resulting in an image consisting only of visible “edges.” We then “subtract” corresponding pixels in consecutive edge-only images (from the same stream). This removes stationary objects, leaving an image in which the edges signify

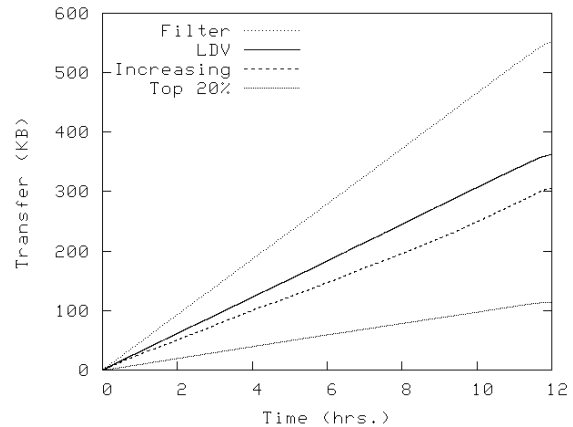


Figure 4. The amount of data transferred to each point in the system. The “filter” is a part of the LDV package, while the “LDV” is the actual internal representation of the live data. Our clients in this case were interested in areas with the top 20% of congestion, and areas of increasing congestion (i.e. positive derivative).

objects that were in one picture and not the other. Figure 3 illustrates our algorithm. In essence we are doing a very simple calculation of the number of moving objects in an image. By computing the average pixel value of this difference image, we are able to determine the level of traffic at a particular moment in time: a higher mean corresponds to more movement.

Our experiments on real-life images indicate that this simple approach is fairly effective at determining traffic levels correctly with an accuracy of 65-70%. Of course, this value is highly dependent on the amount of noise present in the traffic images, as well as the percentage of the image that actually contains the road. We obtain more accurate results when a larger portion of the camera is focused on the traffic itself.

As described in Section 3, we instantiated the LDV with parameters specific to traffic. In our case, that meant a selector statement recognized by the JMS message broker to start up the traffic sensors. We used a value of three minutes for T , since this was the approximate time between image updates on the website. As previously mentioned, we had two clients, one concerned with areas containing the most traffic and another interested in areas of increasing traffic. 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 ones, meaning they subscribed to automatic updates from the LDV.

Figure 4 quantifies the data passing through the system in a 12-hour period. We monitored the three areas of the system 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 error values from the sensors did not make their way into the LDV. The filter also deals with out-of-order messages; however, this feature was not exercised by these simulations. 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. Results computed for the *number of messages*, rather than the total byte count, tell a similar story: with our nominal filtering, the traffic client in both applications received, on average, 50% of the messages 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 two special challenges.

- *Source Scalability*: The LDV must be capable of scaling to a very large number ($O(10,000)$ and above) of potential data sources, especially for applications such as environmental monitoring where the LDV is calculated from the deployment of a pervasive sensor network.
- *Many Packet Arrival and Timer Events*: The LDV must not only be able to support high data throughput from individual streams, but 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.

We now survey our ongoing work at addressing these challenges within “industry-standard” runtime environments.

5.1. An LDV Implementation Based on J2EE

The J2EE programming model **Error! Reference source not found.** defines one of the more popular runtime architectures for component-based enterprise applications. Enterprise Java Beans (EJB) define one of the more commonly used J2EE approaches for developing modular application logic, with the EJBs representing infrastructure-agnostic business logic and a server “container” providing features such as consistency, scalability and concurrency within a distributed runtime infrastructure. The EJB model is presently geared towards applications such as e-commerce, CRM, and supply-chain management that consume only traditional “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.

Figure 5 shows a logical diagram of how the various components of an LDV can be mapped into various types of EJBs. In our model, a client invocation of an LDV causes the server to logically maintain its own version of

the LDV. Of course, multiple clients may connect to multiple instances of an LDV.

Our implementation of the LDV model assumes that an available data stream is mapped onto a “topic” publisher, so that data elements from the same source arrive as distinct messages with a common “stream ID” field. A Message Driven Bean (MDB) is parameterized by a topic, which identifies the type (SSpec) of the LDV data sources, and a selector that filters the streams to ensure that the message attributes satisfy a predicate (MSpec). The MDB also timestamps each incoming message. The unexpired streams structure is implemented as a Stateful Session Bean (SSB) that receives a) incoming data elements (messages) from the MDB and b) timer-based expiration or staleness events. A specific instance of an SSB is associated with a fixed value of T . The SSB then generates its own events (such as a “new data value arrived,” or “existing data value expired”) for the intra-stream operators (f_1), which in turn generate appropriate events for the inter-stream operators (f_2). The operators f_1 and f_2 , as well as the Specification Operator (Spec), are implemented as stateful session beans as well. The f_1 , f_2 and Spec beans thus act as listeners, waiting for events from their respective upstream senders. An LDV client only interfaces with an abstract “Live Data Bean,” which is in reality the Spec SSB, and is shielded from the underlying implementation details and the connections between the MDB and the various SSBs.

Figure 5 presents only one of several possible approaches to optimizing an LDV implementation across multiple clients. Note that in this approach, there is a single LDV (with one value for T) that is shared by multiple application clients. Our initial J2EE-based implementation follows this architecture, in the belief that multiple clients of the same type of dynamic data are likely to expect or tolerate identical notions of liveness, and differ only in the choice of operators. Thus, for our traffic scenario, the applications differ in the interpretation of the data, not on the window over which freshness is defined. Alternative implementations (e.g., a separate LDV for each client) are possible, if the clients exhibit other forms of heterogeneity. We leave the investigation and detailed evaluation of various alternative J2EE-based runtime architectures for LDV to a future paper.

5.2. Scalability and Consistency Challenges

There are many interesting design challenges related

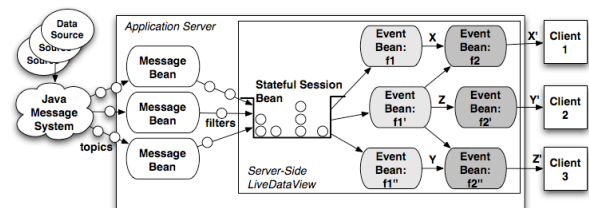


Figure 5. Server-side implementation of a Live Data View using J2EE

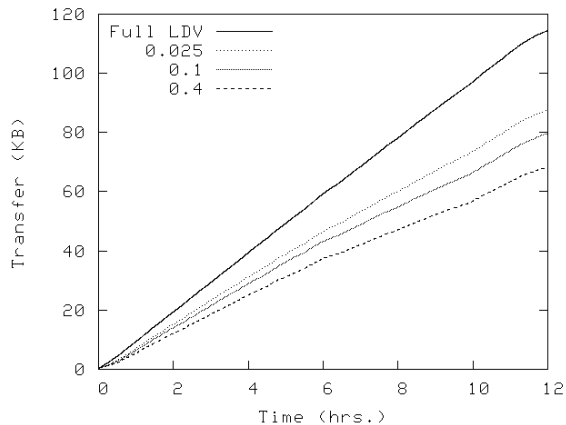


Figure 6: Data sent to client under varying threshold values. In each case, clients only wanted updates if f_2 differences exceeded the given threshold. The "Full LDV" is for a client who wants all f_2 results, regardless of whether or not there was change.

to scalability and consistency, especially as LDVs evolve to handle large numbers of streams or clients. Within J2EE, the use of MDBs makes the process of JMS selection and filtering fairly scalable, since the container is free to create multiple concurrent instances of a particular MDB to handle changes in the stream load. Of course, the use of multiple (possibly distributed) MDB instances is only possible due to the looseness of the basic LDV model—since an LDV assumes no fine-grained clock synchronization, different MDBs may timestamp incoming messages independently. Slight differences in (real) clock values are accommodated by the notion of currency parameterized by τ .

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 potentially reduce the traffic volume between an LDV server and the client. For example, Figure 6 plots the message 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. We can clearly see that, allowing for even small divergence between the true and client views, this can reduce the traffic volume significantly, and thus increase the scalability of LDVs. LDV targets a category of monitoring applications, where changes in the environmental state (such as temperature readings or traffic congestion levels) often occur relatively slowly and smoothly. In such environments, we expect that even small, non-zero values of the tolerance threshold D help filter out transients and will significantly improve the system scalability.

Moreover, defining tolerance over an LDV in terms of a divergence measure introduces a new time-based weak-consistency model across multiple clients. Intuitively, if two clients C_1 and C_2 define their tolerance

thresholds D_1 and D_2 in terms of a “norm” $\|\cdot\|$, then we can guarantee that the divergence between the two client views is $< \|D_1 + D_2\|$. We believe that research into this novel concept of “consistency over time-derived views” can result in interesting consistency semantics that are complementary to the extant work [12][13], which supports multiple consistency models over either data from individual streams or over time-independent database relations.

Finally, the possibility of multiple distributed instances of the LDV generates additional consistency and scalability issues. A scalable server-side runtime might create “replicas” of a “master” LDV, where the MDBs feed a specific LDV instance, and additional instances mirror the contents of the LDV. Different clients, and their associated f_1 and f_2 operators, may then be fed by different LDV instances for better load distribution. A “fully consistent” model would require the master MDB to distribute every message and timer-based event to all replicas (especially when there is non-negligible distribution latency). However, it is possible that the distribution burden would be significantly reduced (and the system scalability enhanced) if the replicas were allowed to diverge up to a threshold D (under some metric). For example, if the master knew the f_1 and f_2 operators used by a replica, it could choose not to forward those stream elements that have no impact on the eventual derived state. In general, developing appropriate metrics and consistency mechanisms for bounding the “time-derived views” derived from distributed LDV replicas is an open research problem.

6. Related Work

Applications centered on monitoring the physical environment are prevalent in many domains, including the wireless sensor community. Mainwaring *et al.* [14] presents a reference architecture that covers habitat monitoring of microclimates, where distributed sensors generate data streams that are processed by backend servers. LDV would be useful in this domain as a tool to aid in the construction of clients that use the sensor information. Network monitoring is also an 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 [1] and Aurora [2] 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 an application-level abstraction of dynamic data as a set of streams. LDV provides a simplified view of dynamic data. 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 [15] and SMILE [5]. 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 [1].

7. Conclusions and Future Work

In this paper, we presented the Live Data Views model for representing dynamic state derived from data streams. We provided an overview of the model, described how the model can be incorporated into an application, and then demonstrated the efficacy of our approach through a proof-of-concept Java prototype that reports traffic conditions based on highway images taken by cameras that are publicly accessible on the Internet. We evaluated the overhead of the LDV approach for that specific application. We then presented the design and implementation of the server-side components necessary to support LDVs on clients. We expect that a solution similar to LDV will become a major feature in future versions of enterprise programming frameworks that need to support dynamic data.

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 function 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] 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, 1st Biennial Conference on Innovative Data Systems Research (CIDR) 2003
- [2] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. In VLDB Journal (12)2: 120-139, Aug 2003
- [3] S. Raman, S. McCanne: A Model, Analysis, and Protocol Framework for Soft State-Based Communication. Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), 1999, pp. 15-25.
- [4] P. Eugster, P. Felber, R. Guerraoui, A. Kermarrec. The many faces of publish/subscribe. ACM Comput. Surv. 35(2): 114-131 (2003)
- [5] Y. Jin and R. Strom. Relational Subscription Middleware for Internet-Scale Publish-Subscribe. Proceedings of the 2nd International Workshop on Distributed Event-based Systems (DEBS), 2003, pp 1-8.
- [6] <http://java.sun.com/products/jms/>
- [7] Connecticut Department of Transportation, <http://www.conndot.ct.gov>
- [8] Washington State Department of Transportation, <http://www.wsdot.wa.gov>
- [9] A. Bevilacqua and M. Roffilli, Robust denoising and moving shadows detection in traffic scenes, IEEE Conference on Computing Vision and Pattern Recognition. December 2001.
- [10] W. Frei and C.C. Chen. Fast boundary detection: A Generalization and a New Algorithm. IEEE Transactions in Computing, vol. 26, no. 10, pp. 988-98, Oct 1977.
- [11] <http://java.sun.com/j2ee>
- [12] H. Yu and A. Vahdat, Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services, ACM Transactions on Computer Systems, Vol. 20, No. 3, Aug 2002, pp 239-282.
- [13] 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
- [14] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring, Proceedings of the 1st ACM International Workshop on Wireless Snesor Networks and Applications (WSNA) 2002, pp 88-97
- [15] M. K. Aguilera, R. Strom, D. Sturman, M. Astley, T. Chandra. Matching Events in a Content-Based Subscription Systems, Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC), 1999, pp 53-61.
- [16] C. Olston, J. Jiang, and J. Widom. Adaptive Filters for Continuous Queries over Distributed Data Streams, Proc. 2003 ACM SIGMOD, pp 563-74