2018 IEEE 34th International Conference on Data Engineering

Actor-Oriented Database Systems

Philip A. Bernstein

Microsoft Corporation
One Microsoft Way, Redmond, WA, 98052, U.S.A.
philbe@microsoft.com

Abstract—**We present the vision of an actor-oriented database. Its goal is to integrate database abstractions into an actor-oriented programming language for interactive, stateful, scalable, distributed applications that use cloud storage.**

I. Introduction

Many of today's interactive, stateful, server applications are processor-intensive and must be scalable and elastic. Hence, they are usually implemented using middle-tier servers backed by cloud storage, rather than using stored procedures in a database system. This has two benefits. First, the mid-tier can offload the storage back-end by caching data that is frequently accessed. This enables the system to scale elastically by adding or removing inexpensive middle-tier servers. Since mid-tier servers can access all of cloud storage, this elastic scaling does not need data migration. Second, it enables the system to scale computation and memory by adding inexpensive mid-tier servers, independently of adding I/O. This is beneficial to mid-tier applications that render images, compute over large graphs, process high-volume message streams, or perform other computation-intensive activities.

Mid-tier applications are often object-oriented, because they involve interactions with large numbers of objects in the real world. Examples include multi-player games, social networking, mobile computing, telemetry, and Internet of Things.

Since real-world objects are independent of each other, the software objects that model them do not share state and communicate via asynchronous messages. Such objects are called actors. Therefore, the actor programming model is popular for implementing these types of applications. There are dozens of programming frameworks for building actor applications, such as Akka [1], Erlang [2], Orbit [9], and Orleans[10].

In these types of applications, each actor typically has state that describes the real-world objects it models and that survives across multiple calls to the object. This data is not always stored persistently, for example, because it is device state that changes frequently or because it models a real-time session that disappears when its participants disconnect. Even if the actor's state is stored persistently, its freshest state may be in main memory, which is only periodically written to persistent storage.

In summary, actor applications manage large numbers of long-lived stateful objects distributed across many servers. Thus, they are effectively database applications, even though they are rarely thought of that way. However, actor programming frameworks offer few, if any database abstractions, such as indexing, transactions, streams, replication, geo-distribution, and queries. This is an opportunity for the database field.

For the past four years, we have been working with developers and users of Orleans, an open-source .NET programming framework for scalable, distributed actor-based applications [4][6]. Orleans application developers have asked for database abstractions, to improve programmer productivity and application robustness. We have therefore been adding them to Orleans and summarize here what we have learned.

We use the term *actor-oriented database* (AODB) to describe an actor framework that incorporates database abstractions. It has three properties that distinguish it from other types of database systems: it scales out elastically to a large number of servers, it is agnostic to the storage system which it treats as a plug-in, and it is compatible with the actor framework's programming model [6].

II. Related Work

There have been many previous approaches to incorporating database functionality into object-oriented systems: object-oriented databases (OODBs), persistent programming languages, distributed object systems, and object-relational mappers (ORMs). Like an AODB, they all strive to avoid an impedance mismatch between the database interface and programming language. However, except for ORMs, they all were developed with a customized server-attached storage system. By contrast, an AODB must work with a broad range of cloud storage systems. They differ from AODBs in a variety of other ways: most OODBs were built for computer-aided design, some persistent programming languages offer persistence-by-reachability, and most ORMs target SQL for their storage. As far as we know, none of them focus on scalable interactive services.

Other related technologies are enterprise software platforms (e.g., Java EE), mid-tier cache managers (e.g., memcached), and graph databases. There is much to learn and borrow from all these technologies. But the details and combination of features needed by AODBs is different from all of them

III. Components

A. Identity and Location Transparency

To invoke a method on an actor, the actor needs an identity. Ideally, the identity is location-transparent, which simplifies load-balancing and fault tolerance. It might be long-lived, much like the primary key of a database table. Or it might be transient and a new one minted whenever an actor is activated.

B. Transactions

In an AODB, a transaction spans multiple actors that in general run on different servers and independently write their state to cloud storage. Therefore, for atomicity and durability, two-phase commit is required. If a write to cloud storage takes 20ms (a typical number), then the maximum throughput on a

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00010

write-hot actor is 25 transactions/second. We have developed a protocol that uses early lock release to enable batching, which can increase throughput by an order of magnitude [8]. Although today's cloud storage systems do not offer logging as a service, if they did, as in [3], then other optimizations might be possible.

C. Replication and Geo-distribution

There are two natural programming models for replicated actors: primary-copy and multi-master. In primary-copy, each distinct actor has at most one running instance. If the application is geo-distributed, this single-instancing must be guaranteed world-wide. If an actor has replicas running as warm or hot standbys, then after an actor fails, a leader-election algorithm chooses another actor as primary. Or the actor might be made highly available by using replicated storage to manage its state, a capability widely supported by cloud storage services. The latter is our current solution in Orleans.

In a multi-master setting, many updatable copies of an actor may be running. This is mainly useful for geo-distribution, with copies running in different datacenters. The system should offer a well-defined programming model for actor-state synchronization. In our solution [5], the actor state is eventually linearizable, by which we mean that updates are eventually applied to all copies in the same order and actor state is always a prefix of the linearized sequence. An application can also read the latest linearizable state it knows about, with all of the application's subsequent updates applied. This state might never exist in the linearizable sequence but is nevertheless useful to some apps.

In both the multi-master and primary-copy setting, geo-distributed transactions could be supported, which is a subject of our current research.

D. Indexing

In addition to accessing actors based on their identity, it is often useful to access them based on other member variables [7]. For example, for a Player class in an on-line game, it may be useful to find all players at a given skill level, e.g., to pair them up in a game. This may be required whether or not the actors are stored persistently and, if they are stored persistently, whether or not the underlying storage system supports indexing. Developers have also asked to index only the actors that are currently executing, e.g., to notify all players at a given game-location of a nearby battle.

E. Streams

In addition to performing actor-to-actor message communication, applications often include reactive computations that respond to events streams. The computations might be standing queries or imperative code. The streams can be delivered over a variety of underlying messaging and queueing technologies, so the stream source's transport should be a plug-in. In addition to wrapping a transport in a common API, the plug-in could have logic to aggregate events or filter events within a stream, to merge or fork a stream, or to do other custom processing.

Ideally, streams should be fault-tolerant. If an actor that is processing a stream fails, it should be able to recover and continue processing at the point it left off, to ensure at-least-once, at-most-once, or exactly-once execution, as desired.

Orleans offers an extensible pub-sub system [11]. It is a push model, where an actor registers a callback method with a stream source, after which it receives a call for every relevant event. The stream source's transport is a plug-in and can pre-process events to aggregate or filter them. The actor that processes events could be a complex event processing engine, such as Trill [12], or custom application code.

F. Queries

By treating actors as a special kind of object, an object-oriented query language can be applied to an actor system. If an actor database is cast as a graph, where actors are nodes and references from one actor to another are edges, then queries could be path-oriented, such as regular expressions or SPARQL.

Since storage is a plug-in, the query optimization problem is similar to that of object-to-relational mapping systems. However, there are several differences. Some actors may have fresher state in memory than in storage. Some actors are not mapped to storage, yet they still need to be queryable. Moreover, like indexing, query processing sometimes should be applied only to running actors.

Often, actors are dynamically assigned to servers with no fixed affinity. This complicates query processing since the data relevant to a query could be on every server. It also complicates materialized view maintenance, since a view may need to receive updates to actors running on every server.

IV. Conclusion

We have made many tradeoffs in selecting database features for Orleans. There is much room to explore other alternatives.

References

[1] Akka documentation, http://akka.io/docs/

[2] Armstrong, J., "Erlang," *CACM* 53, 9 (2010), pp. 68–75.

[3] Balakrishnan, M., D. Malkhi, J.D. Davis, V. Prabhakaran, M. Wei, T. Wobber: CORFU: A distributed shared log. ACM Trans. Comput. Syst. 31(4): 10:1-10:24 (2013)

[4] Bernstein, P.A., S. Bykov, A. Geller, G. Kliot, J. Thelin *Orleans: Distributed Virtual Actors for Programmability and Scalability*, MSR-TR-2014-14, http://research.microsoft.com.

[5] Bernstein, P.A., S. Burckhardt, S. Bykov, N. Crooks, J.M. Faleiro, G. Kliot, A. Kumbhare, M.R. Rahman, V. Shah, A. Szekeres, J. Thelin: Geo-distribution of actor-based services. PACMPL 1(OOPSLA): 107:1-107:26 (2017)

[6] Bernstein, P.A., S. Bykov: Developing Cloud Services Using the Orleans Virtual Actor Model. IEEE Internet Computing 20(5): 71-75 (2016)

[7] Bernstein, P.A., M. Dashti, T. Kiefer, D. Maier: Indexing in an Actor-Oriented Database. CIDR 2017

[8] Eldeeb, T. and P.A. Bernstein, "Transactions for Distributed Actors," MSR-TR-2016-1001, http://research.microsoft.com/.

[9] Orbit, https://github.com/orbit

[10] Orleans, http://dotnet.github.io/orleans

[11] Orleans streams, http://dotnet.github.io/orleans/Documentation/Orleans-Streams/index.htm

[12] Trill, https://www.microsoft.com/en-us/research/project/trill