Actors and Publish/Subscribe: An Efficient Approach to Scalable Distribution in Data Centers

Dominik Charousset
HAW Hamburg
dcharousset@acm.org

Thomas C. Schmidt
HAW Hamburg
t.schmidt@ieee.org

Matthias Wählisch
Freie Universität Berlin
waehlisch@ieee.org

ABSTRACT

Data center applications are required to be fault-tolerant and self-healing, and at the same time to scale dynamically with the number of available hardware resources. Highly efficient task distribution is crucial for such services that require low latency and high availability. This paper introduces pub/sub actors as a paradigm to build distributed data center applications without a single point of failure. Our approach does not actively distribute tasks, but uses group communication and an orchestration protocol. Requests are received by a group of potential servers, but only processed by one of them. We present a key-value store using `libcppa` as a case study of promising performance.

Categories and Subject Descriptors

C.2.4 [**Computer-communication networks**]: Distributed Systems —*Distributed databases*; D.3.3 [**Programming languages**]: Language Constructs and Features—*Concurrent programming structures*

Keywords

actor model, C++, key-value store

1. INTRODUCTION

The traditional actor model [3] defines a message-oriented programming paradigm which allows for a strong coupling of (sub) systems by link states. This approach naturally fits to master-slave relationships and has proven useful in practice to build fault-tolerant, hierarchical distributed systems. Erlang-like failure models for actors has been used for reliable, highly available, and self-healing applications. It may be seen as a natural candidate for in-data center development. However, strictly hierarchical systems exhibit limited flexibility and high communication overhead in distributing tasks by a master to its slaves. Seamless call delegation and a contacting of unknowns are not covered by this model.

Loosely coupled systems can have a significant advantages in low-latency applications, if they rely on orchestration and scalable network facilities rather than centralised task management. By using publish/subscribe, one can forgo central task management and avoid most of its communication overhead. However, the traditional actor model does not provide a publish/subscribe semantic.

In this paper, we present a loosely coupled actor paradigm as an enhanced approach to data center programming (§ 2). Our work is based on `libcppa` (`http://libcppa.org`), a C++ library for actors that extends the basic model by a publish/subscribe communication layer. This software infrastructure is evaluated in the context of `psaDB`, a prototypic key-value store for data centers (§ 3).

2. `psaDB`: A KEY-VALUE STORE

As a characteristic data center application, we present `psaDB` (publish/subscribe & actors DataBase), a key-value store that serves requests from a distributed server pool. The traditional actor model defines only a one-to-one send primitive using actor addresses. We have enhanced this model to allow for one-to-many communication in `libcppa`, an open source implementation of the actor model for C++. It allows actors to join and leave groups, as well as send messages to and receive from (joined) groups. This paves the path to new use cases, such as low-latency applications without central task management.

`psaDB` delegates each request to a group of servers from a large server infrastructure. We have separated this delegation process into a static, "off-line" step and a dynamic adaptation to runtime configuration. Initially, the system is divided into a predefined number of groups. Each group serves a fixed key range and can be transparently addressed by a group address `g`.

Each client receives a list of all group identifiers at configuration time. At each request, the client independently maps the key to a the corresponding group using a hash function. This first step delegates a call to a subsystem without any communication overhead.

Each group of servers consists of an (undisclosed) master that internally assures consistency and coordination, as well as any number of replicas. For a specific deployment scenario, the master partitions the key space of its group evenly among all replicas and itself. This is the second step of the call delegation. Only the server that is currently responsible for a given key responds to a request. For write requests, each replica performs the write operation, but only the currently responsible replica acknowledges the message. `psaDB` assumes a reliable, FIFO-ordered multicast transport. Whenever a new replica joins or an existing replica fails, the master redistributes the key space. If the master fails, the replicas start an election of a new master. To detect failing servers, each replica (actor) monitors each other replica (actor) in its group.

psaDB scales in two ways, (a) the database can configure many groups to divide data as well as workload across many subsystems, and (b) administrators can add replicas to subsystems at runtime to narrow the key range each server has to respond to. As shown in Figure 1, put and get operation always need exactly two messages, a request message that is sent from the client to the subsystem (group of servers), and a response/acknowledge message that is returned. Since this approach uses group messages of a specific anycast-style communication, it assumes an efficient group communication layer in order to scale up to a large number of replicas.

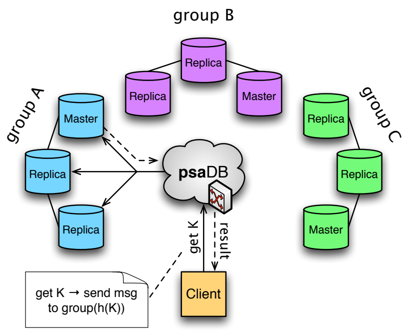

Figure 1: Distribution Concept of psaDB

This paradigm of direct group access can be generalized to any stateless request/response protocol such as HTTP and can be applied to distributed computing on clusters, e.g., Map/Reduce, as well.

3. PROTOTYPE AND EVALUATION

We have implemented psaDB, a proof-of-concept key/value store. It supports get, put, and scan operations and is written in C++ with the actor library libcppa.

On the client side, psaDB is represented by a single, local actor that forwards all requests transparently to the corresponding groups. On the server side, the actor model provides all tools necessary to build reliable systems based on actor monitoring. The complexity during development is minimized, since the subsystems remain isolated.

For early testing and evaluation, we have deployed our prototype on three groups consisting of one master and two replicas each connected via a 1GB/s switched Ethernet network. The group service convergence layer was native IP Multicast. To evaluate the performance of our setup, we have used workloads A–E of the Yahoo! Cloud Serving Benchmark (YCSB) [1], an industry-standard benchmarking tool. Workload D uses a "latest" read pattern, i.e., it accesses the last inserted or updated values, whereas all other workloads use a Zipfian access pattern.

Table 1 shows the average latency as well as the 95 and 99 Percentile for the benchmark. The database was preloaded with 100,000 elements for each workload. Each element consists of ten 1 kB sized values. The benchmark illustrates the bare overhead of psaDB itself, as our prototype uses only in-memory storage with $O(log\ n)$ complexity for accessing elements. Therefore, read and write operations performed at almost identical speed. The latency is likely to increase slightly, when data access involves IO operations on the physical hard drive.

	Operations	Average	95 %	99 %
A	50 % Read	748 μs	1 ms	2 ms
	50 % Update	725 μs	1 ms	1 ms
B	95 % Read	738 μs	1 ms	2 ms
	5 % Update	713 μs	1 ms	1 ms
C	100 % Read	760 μs	1 ms	2 ms
D	95 % Read *	731 μs	1 ms	2 ms
	5 % Insert	1055 μs	2 ms	2 ms
E	95 % Scan	24969 μs	32 ms	36 ms
	5 % Insert	10857 μs	16 ms	19 ms

* D has a latest read pattern, default is Zipfian

Table 1: psaDB performance for several workloads

DB	95 % A	95 % B	95 % E
HyperDex	3 ms	3 ms	5 ms
MongoDB	4 ms	6 ms	120 ms
Cassandra	16 ms	20 ms	80 ms

Table 2: Benchmark results of Escriva et al. [2] for YCSB workloads A, B, and E

For comparison we show the benchmark results of Escriva et al. [2] in Table 2 from their work on HyperDex in a 14 nodes setup. The Percentiles show the maximum time for all operations in a certain workload. MongoDB and Cassandra are freely available, open source database implementations. Note that Escriva et al. did not show detailed results for workloads C and D. However, results for workloads C and D should correspond to the results of workload B.

Despite the limited compatibility of measurements, we diagnose that our early prototype runs on the same timescale as mature systems. psaDB's only flaw in performance compared to HyperDex is its scan performance, as psaDB does not have a coordinator that manages a global state. Due to the fully distributed architecture of psaDB, a client requesting a key K and its next x successors (workload E, usage example: threaded conversations) has to send its query to all groups and then to combine and truncate the individual results. In following a fully distributed approach, psaDB admits much higher flexibility in deployment, omits any single point of failure, and – without data state at a controller – avoids inconsistency by design.

4. REFERENCES

[1] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the 1st ACM SoCC* (New York, NY, USA, 2010), ACM, pp. 143–154.

[2] Escriva, R., Wong, B., and Sirer, E. G. HyperDex: A Distributed, Searchable Key-Value Store. In *Proc. of the ACM SIGCOMM* (New York, NY, USA, 2012), ACM, pp. 25–36.

[3] Hewitt, C., Bishop, P., and Steiger, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. of the 3rd IJCAI* (San Francisco, CA, USA, 1973), Morgan Kaufmann Publishers Inc., pp. 235–245.