DEVELOPMENT OF A NEW JAVA CHANNEL ACCESS LIBRARY JCAL

Hiroyuki Sako#, Japan Atomic Energy Agency, Tokai, Japan
Hiroshi Ikeda, Visible Information Center Inc., Tokai, Japan

Abstract

Java channel access library JCA (Java Channel Access library) has been widely used for device control applications in Java. Especially for high-level applications in the J-PARC linac and RCS (Rapid-Cycling Synchrotron) control systems, which are unified in Java, the pure version of JCA implementation (CAJ) is desirable. However, JCA and CAJ have instability problems and vulnerability of the codes. To overcome the issues, a new compact Java channel access library, JCAL (Java Channel Access Light library) has been developed. A special care is taken to design the code architecture in order to keep thread safety and code robustness. The main part of the library is designed to work in a single thread, with the other threads for the monitor and call-back functions of Channel Access. By adopting such a simple design, robustness and stability is realized. An adapter library for JCA API, JCA-JCAL, has been also implemented for convenience to plug in JCAL to existing Java applications with JCA or CAJ. Bench mark tests have been carried out and compared to JCA, which show comparable performance.

INTRODUCTION

JCA (Java Channel Access library) is a channel access library for Java applications developed by Cosylab [1]. The library consists of API (Application Program Interface), and its implementations. There are pure Java implementation, CAJ, and JNI implementation, JCA-JNI. The latter is more stable and has been widely used in EPICS control systems such as J-PARC and SNS.

The JCA library, especially the pure Java implementation CAJ (Channel Access in Java) [2] is desirable for the J-PARC linac and RCS control system whose presentation layer is based on Java. However, JCA has vulnerability in implementations as described below.

Thread-Safety Violation

It is hard to predict what kinds of problems happen if a multi-threaded application violates thread-safety, since the effect spreads all over the codes. Thread-safety violation can be found easily if the application contains apparent insufficient or inconsistent synchronizations, or well-known anti-patterns such as invoking the wait method without a condition loop, calling anonymous methods with holding locks. Actually, some of them appear in the API of JCA and JNI and CAJ implementations. However, generally it is hard to find broken thread safety and repair it perfectly, since a thread can access many parts of the application, all of which must be followed by an investigator must follow.

Vulnerable Internal Structures

Broken encapsulation is a fundamental defect from the object-oriented programming point of view, for instance, returning mutable fields without defensive copy, escaping the reference **this** directly/indirectly in the constructor, which appear in the API of JCA and JNI and CAJ implementations.

Strong dependencies among different parts of codes make it hard to maintain and improve the library. Especially, interdependencies among packages must be avoided.

JCA API Specific Problems

The API contains too concrete implementations, which is not necessary for users. It should be more abstract to reduce its conceptual weight, in other words, to reduce amount of users' knowledge of the codes. To be worse, since an implementation part of some API classes there have defects, their derived classes are forced to inherit the defects.

For example, the abstract class **DBR** defined in the API represents a fundamental data structure. It is defined as an abstract class instead of interface, and implementations of some functions make **DBR** neither immutable nor thread-safe. Thus **DBR** is hard to handle in the context of multi-thread. Also, its implementation details expose its internal field, which is a fundamental defect in object-oriented programming. Moreover, there are too many subclasses of **DBR** included in the API, due to “combinatorial explosion” in combinations of base types and attribute types. Also some subclasses are forced to implement an interface which is useless for the library users, due to implementation details. Since these subclasses all inherit the properties of **DBR**, they are neither immutable nor thread-safe, and expose their internal information. Even though these defects exist, implementations of API are forced to use **DBR** and its subclasses, and require extra efforts to handle them in the multi-threaded context. Even worse, these defects will never be repaired in future releases, to keep the backward compatibility.

The API also includes a non-standard naming style which confuses Java developers.

There are magic numbers hardwired in the API which are used to construct and parse network packets. This means that the implementation details are included in API and exposed, which reduce flexibility and extensibility of the API.

The class QueuedEventDispatcher, implementing the interface EventDispatcher, includes the following anti-patterns and defects. It starts a thread in the constructor, which exposes an incomplete instance. A daemon attribute of the dispatch thread is set, which might stop a

#hiroyuki.sako@j-parc.jp

critical process, for instance, writing data to a file. It invokes "wait" method without a condition loop. Its synchronization policy of listeners to fire is vague.

CAJ Specific Problems

Using **finalizers** is almost anti-pattern, because it is not clear whether and when VM executes the **finalizer**. Moreover, implementing **finalizer** deteriorates the performance.

Since validity check for data received via network is insufficient, the application using the library crashes when it has received some invalid packets.

It is ambiguous which methods are thread-safe in **Channel**, because its implemented class has too many responsibilities with insufficient synchronizations, and it includes the anti-pattern of escaping the reference **this** from its constructor.

There is no document for the synchronization policy which variables are needed to guard by which locks.

The Leader/Followers pattern is adopted in CAJ which uses a thread pool to handle received tasks to reduce overhead of passing the tasks from one thread to another. On the other hand, this pattern complicates the codes, which might break thread-safe. It is in general difficult to use a multithreaded system effectively in the client-side applications, because there are control flows to request tasks from the user-level to the network-level, opposite to control flows to handle responses as events from the network-level to the user-level. It also makes the library hard to use if callbacks of the events are invoked by several threads. Therefore it is doubtful whether using this pattern in CAJ is necessary.

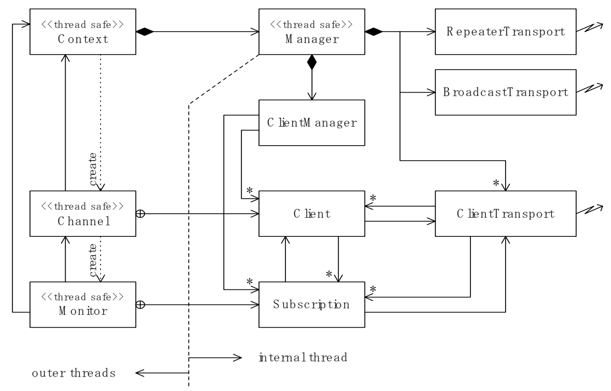

Figure 1: JCAL architecture.

IMPLEMENTATION

To overcome the above problems of JCA, we designed and constructed a new Java channel access library, JCAL (Java Channel Access Light library) [3].

Fig. 1 shows the schematic view of JCAL core structure. JCAL works in a single thread, which is called the internal thread. Only **Context**, **Channel**, and **Monitor** in API and **Manager** are cross-accessed from other threads (outer threads) and implemented thread safe.

Table 1 describes each class in Fig. 1. **Channel** and **Client** are different views of the same entity, where **Client** is defined as an inner class of **Channel**. There is a similar relation between **Monitor** and **Subscription**. **ClientManager** holds instances of **Client** and **Subscription, RepeaterTransport**, **BroadcastTransport** and **ClientTransport** represent abstract communications using TCP or UDP sockets. **RepeaterTransport** listens for beacons from servers, by registering a **Repeater**. **Repeater** is an independent application that allows clients and servers in a host to share a UDP port. JCAL does not include **Repeater**. **BroadcastTransport** sends broadcast packets and receives packets to search for channels. **ClientTransport** exchanges data with a server for channels and monitors, which dynamically bundles channels and monitors into a single TCP connection (which is called Virtual Circuit) if they are connected to the same server in a same priority. This complex procedure for **ClientTransport** comes is requires by the channel access protocol.

Manager holds not only references to the data used by the internal thread, but also the main logic of the internal thread and implementations of the **Transport** classes.

We have also developed a convenient adaptor library for JCA-API with JCAL, JCA-JCAL. By just setting “jp.go.jaea.jcal.jca.JcalContext" to JCALibrary.createContext method instead of values for JCA-JNI and JCA-CAJ, an application uses JCALJ. Parameters of JCA-JCAL can be set in "JCALibrary.properties" configuration file that is for JCA.

Table 1: Classes in JCAL

Class	Functions
Context	Library environment
Channel	EPICS channel (API)
Monitor	EPICS monitor (API)
Manager	Manager to control the internal thread
ClientManager	Manager for Client and Subscription
Client	Inner class of Channel, accessed only by the internal thread
Subscription	Inner class of Monitor, accessed only by the internal thread
RepeaterTransport	Communication with Repeater (UDP)
BroadcastTransport	Communication in broadcast (UDP)
ClientTransport	Communication with server (TCP/IP)

Benchmark Tests

To examine the performance of JCAL, we have carried out benchmark tests compared with JCA-JNI, JCA-CAJ, and JCA-JCAL. For JCA and CAJ, we used JCA-2.3.2, and CAJ-1.1.5b. The number of channels is varied from 4 to 4000. We have iterated the same test 10 times and took average time except for the first iteration, to exclude time for loading classes. To measure the speed accurately, we waited 10 seconds before each test to avoid influence of a previous test, and we called garbage collections before each test, so that it is not executed during the test.

We have performed tests in two environments; a software IOC (Input Output Controller) and IOCs for beam monitors in the J-PARC control system. Fig. 2 shows the duration of the "connection" test as a function of the number of channels for the software IOC. The order of durations (from shortest to longest) is JCAL, JCA-JNI (single-threaded), JCA-JCAL, JCA (multi-threaded), and CAJ.

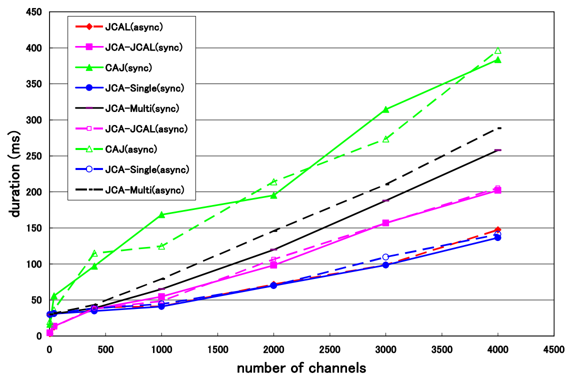

Figure 2: A benchmark test for "connection" in the software IOC.

Fig. 3 shows the duration of the "get" test as a function of the number of channels for the software IOC. The order of durations (from shortest to longest) is CAJ, JNI (single-threaded), JCAL, JCA-JCAL and JCA-JNI (multi-threaded).

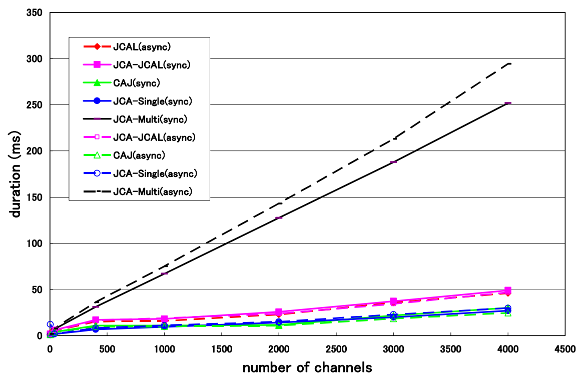

Figure 3: A benchmark test for "get" for the software IOC.

Fig. 4 shows the duration of the "put" test as a function of the number of channels for the software IOC. The order of durations (from shortest to longest) is JCA-JCAL, JCAL, CAJ, JCA-JNI (single-threaded) and JCA-JNI (multi-threaded).

Tables 2 and 3 show summary of duration time per channel with the software IOC and beam monitor IOCs, respectively. It takes much longer time with beam monitor IOCs than with the software IOC. JCAL is comparable with JCA-JNI (single-threaded) and JCA-CAJ. JCA-JNI is the fastest. JCAL is faster than CAJ in connection tests but slightly slower in get and put tests. JCA-JCAL is slower than JCAL, as expected due to overhead by using JCA-API. Multi-threaded JCA-JNI is very slow.

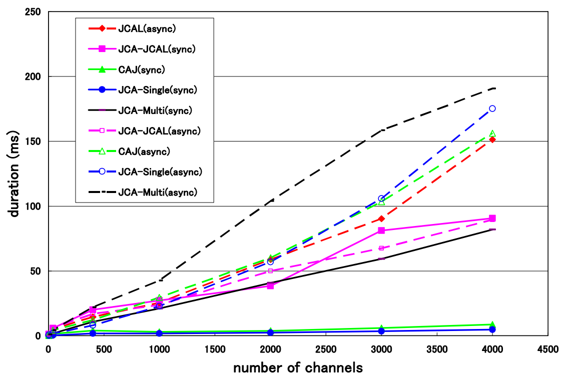

Figure 4: A benchmark test for "put" for the software IOC.

Table 2: Duration per Channel (msec) of Benchmark Tests with Software IOC.

Test	JCAL	JCA-JCAL	CAJ	JNI (single)	JNI (multi)
connect	0.037	0.051	0.099	0.035	0.072
Get	0.011	0.012	0.0061	0.0074	0.074
Put	0.038	0.022	0.029	0.044	0.048

Table 3: Duration per Channel (msec) of Benchmark Tests with Beam Monitor IOCs.

Test	JCAL	JCA-JCAL	CAJ	JNI (single)	JNI (multi)
connect	0.31	0.34	1.01	0.59	0.63
Get	0.040	0.071	0.026	0.031	0.101
Put	0.16	0.37	0.15	0.15	0.18

SUMMARY

A new pure Java channel access library JCAL has been developed for high-level Java application in J-PARC linac and RCS control system. The design and implementation of JCAL is thread safe and stable. Bench mark tests show comparable performance with JCA-JNI and JCA-CAJ implementations.

REFERENCES

[1] http://jca.cosylab.com/
[2] http://caj.cosylab.com/
[3] H. Ikeda, H. Sako, "Development of a New Channel Access Library", WP091, PASJ 2008, Higashihiroshima, Japan.