Porting of Real-Time Publish-Subscribe Middleware to Android

Martin Vajnar, Michal Sojka, Pavel Píša
Czech Technical University in Prague
Technická 2, 121 35 Praha 6, Czech Republic
{vajnamar,sojkam1,pisa}@fel.cvut.cz

Abstract

Real-Time Publish-Subscribe (RTPS) is a protocol, based on UDP/IP, that allows easy and efficient implementation of data-driven distributed real-time applications. The protocol was adopted as an OMG specification and it is intended as an interoperability protocol for applications based on the Data Distribution Service (DDS) API. RTPS is widely used in many industrial applications and it has both commercial and open source implementations. One open-source implementation is called Open Real-Time Ethernet (ORTE) and it is known to work on many platforms including GNU/Linux, Windows, FreeBSD and MacOS. In this paper we describe a new addition to the supported platforms, which is an Android operating system for mobile devices.

We provide an overview of the steps that were needed to port the protocol to the Android platform. We ported the existing native library and added a Java wrapper around it. We comment on the troubles we had and their solutions. We compare the performance of ORTE on Android with other platforms and we present our Android phone application for controlling and monitoring a mobile robot.

1 Introduction

Real-Time applications often need to be distributed over multiple computing nodes. Reasons for that include the distribution of the computing power to the places where it is needed or simplification of the design, management or maintenance of the application. The vital part of all distributed applications is the communication between the application components running on different nodes. In case of real-time applications, the communication is subject to temporal constraints such as deadlines. Applications often need to be fault-tolerant and as such, they must support redundancy in their architecture. Another common requirement is dynamic nature of the application where nodes/components are joined to or removed from the application at run-time. These, often contradicting, requirements make the communication part hard to design and implement. For this reason, people often build their application on top of various communication middleware platforms that handle the communication for them.

Traditional middleware platforms such as CORBA [1] provide transparent access to remote objects by means of remote method calls. When an application invokes a method on a remote object, the middleware automatically serializes the method parameters and sends the request to destination process where the object is located. The computation happens remotely and then the results are sent back. While this simplifies the development of distributed applications a lot, there are many applications that cannot be efficiently implemented on top of this request-response model.

Applications, whose operation is mainly data-driven, meaning that some action/computation is performed when data is ready, would be better served by middleware platforms that seamlessly manage distribution of data from producers to consumers. Such applications are often designed according to Data-Centric Publish-Subscribe (DCPS) model [2], where the middleware creates a notion of a "global data space" that is accessible to all interested applications (see Figure 1). Writing application under such a model brings many advantages. The biggest one being perhaps that the communication requirements are specified by applications in a declarative way and the middleware handles the data exchange automatically based on the declarations.

In this paper we describe recent advances in an implementation of a small DCPS middleware called ORTE [3]. This middleware was ported to the An-

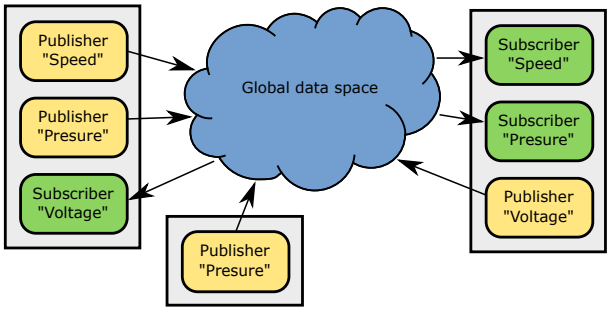

Figure 1: Data-Centric Publish-Subscriber application model.

droid platform, which brings possibility of many interesting applications.

This paper is structured as follows. In Section 2 we describe the DDS API, the Real-Time Publish-Subscribe protocol and the ORTE middleware. Description of how we ported the ORTE to Android follows in Section 3. We evaluate the performance of the Android port in Section 4. Section 5 describes what needs to be done for ORTE to be compliant with the RTPS specification and we conclude in Section 6.

2 DDS, RTPS and ORTE

The popularity of DCPS model resulted in several standardization activities. Data Distribution Service (DDS) for Real-time Systems [2] is an Object Management Group (OMG) standard that specifies an API for applications based on the DCPS model. Applications using this API are portable between different middleware platforms offering this API. The interfaces are specified in OMG Interface Definition Language (IDL) [1, Chapter 7]. This means that the API is defined for many commonly used languages such as C, C++, Python and others[1].

The DDS API allows an application to declare that it produces certain data (identified with a so called topic) or that it wants to receive (subscribe to) such a data. The underlying middleware ensures that the stream of data is properly communicated from one or more producers to one or more subscribers. Besides the basic data exchange between publishers and subscribers, the middleware allows the applications to specify many additional Quality-of-Service (QoS) parameters such as deadlines, reliability level, data durability, etc. This not only simplifies the application design but also allows the middleware to optimize the communication in many ways. The DDS standard specifies only the API, not the underlying mechanisms that implement the data exchange.

The communication protocol that can be used to implement the functionality required by the DDS API is called the Real-Time Publish-Subscribed (RTPS) protocol [4]. The primary goal of RTPS is to provide interoperability between different DDS middleware platforms. This is achieved by defining a minimal set of requirements that all implementations have to satisfy. Besides that, the standard defines many optional advanced features as well as a way for implementing vendor specific extensions of the protocol. The standard defines the protocol in a platform (transport) independent way and then it defines the mapping onto the UDP/IP protocol. The protocol takes advantage of multicast communication when available, but works also in environments without multicast support. Implementations can decide about trade-offs in resource needs. Simple implementations can have a small memory footprint but will need higher network bandwidth. Other implementation can use local memory to highly optimize the communication and thus save network bandwidth.

The RTPS protocol has the following features that make it an interesting option for using in distributed real-time applications.

No single point of failure. Every application has a complete picture of the whole network. Crash of a single application influences only the applications that need data from it.

Redundancy. Multiple publishers can publish the same "topic". If one publisher fails, subscribers will be automatically switched to another one. This is illustrated in Figure 1 where "Presure" information is published by two applications.

Application discovery. Built-in discovery protocol ensures that applications can discover each other as well as the topics they publish or are subscribed to.

Besides several commercial implementations of RTPS protocol, at least two open source implementations exist: OpenDDS [5] and ORTE [3]. OpenDDS seems to be more mature. It implements both the DDS API and the underlying RTPS protocol. It is implemented in C++ and provides bindings for Java. OpenDDS is built on top of the ACE[2] abstraction layer to provide platform portability. OpenDDS also leverages capabilities of CORBA implementation called TAO[3].

[1] See `http://www.omg.org/spec/#Map`

[2] `http://www.theaceorb.com/product/aboutace.html`

[3] `http://www.theaceorb.com/`

ORTE, another open source RTPS implementation, provides its own API instead of the DDS API. The reason is that ORTE development started before the DDS standard was finished. ORTE is implemented in C and includes a small portability layer that allows it to run on many popular platforms including Linux, Windows, MacOS and FreeBSD. Unfortunately, ORTE implements the RTPS protocol according to the draft of the RTPS specification [6] and it would need some changes (see Section 5) in order to be compliant with the current specification. More detailed description of ORTE can be found in [7]. Despite ORTE lack some features of OpenDDS, we deal with it in this paper because we use it in several applications.

3 Porting to Android

Android is one of the most popular and proliferated mobile operating systems. It is running on wide variety of devices ranging from mobile phones and tablets to home media centers and digital cameras. This makes it interesting for soft real-time application developers.

To develop applications for Android, Google offers Android Software Development Kit (SDK) and Android Native Development Kit (NDK). SDK is used for applications written in Java whereas NDK allows to use native C/C++ code in the applications. Android uses Google's implementation of Java Virtual Machine (VM) called Dalvik VM.

Having an RTPS implementation running on Android brings interesting possibilities of using Android devices to control and/or monitor applications that already use RTPS protocol. This was, in fact, our motivation for the porting effort. We have several mobile robots [8] that are built on top of RTPS and we wanted to be able to control the robots via a mobile phone. One such robot is depicted in Figure 2.

3.1 Overview

We considered two possible approaches. One involved writing a pure Java implementation of the RTPS protocol from scratch (as chosen by the PrismTech company [9]), the other was to use a pre-existing Java wrapper, which makes use of the original native ORTE library through the Java Native Interface (JNI) and make it Android-compatible. After thorough consideration we decided to go the Java wrapper way. Mostly because the C code has been in use for quite some time and could be viewed as stable. On the other hand, if we had chosen to write a new implementation of the protocol, it would require extensive testing and would significantly slow our progress.

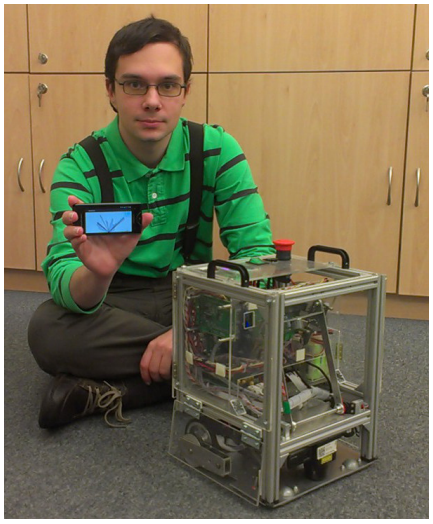

Figure 2: Controling the robot with an Android phone.

In a nutshell, the process of porting was the following.

1. Update Java wrapper that uses Java Native Interface (JNI) and make it Android compatible.
2. Fix bugs that have not demostrated themselves under the Oracle's VM.
3. Add support for Android build system.
4. Make Java version of ORTE Manager application (see Section 5) to overcome problems with execution and termination of native processes.

3.2 Technical Details

In this section we take a closer look at some problems we faced in the process of porting the ORTE library.

3.2.1 64-bit Support

In the Java wrapper pointers to C structures are stored in Java fields for later use. For example each publisher is assigned an instance of ORTEPublication structure upon its creation for future manipulation, the most important being the possibility to send new data.

Java implementations does not have a data type specifically intended for storing pointers like the C

types `intptr_t`, `ptrdiff_t` or `size_t`, so initially we stored pointers in the Java's `int` type, which is 32-bit wide. This works well on current versions of Android, because most devices use 32-bit ARM based CPU, but does not work if used with Oracle's VM on a 64-bit OS. In order to overcome this we considered a possibility of creating `long` fields (64-bit) along with `int` fields and switch between them based on the word length of the system being used on, but it turned out that Java doesn't have a reliable way to determine the native word length. On the Sun/Oracle implementation, there is *sun.arch.data.model* system property used for this purpose, but it is not standardized and Dalvik VM implementation does not support this. Another possibility was to call a C function through JNI to determine the word length at the time of creation of Java objects that need to store native pointers. Calling the JNI function would add an overhead that would not be compensated by performance gain from using shorter fields to store pointers. Based on this we decided to store pointers unconditionally in long fields. This is also the recommended way by Google [10].

3.2.2 Use of WifiLock and MulticastLock

Android makes extensive use of power-saving features. One of them is the Wi-Fi Power Save Polling (PSP) mode [11]. When this mode is entered, the device asks Wi-Fi Access Point to cache all the downlink frames intended for that device. Cached frames could be delivered during a wake-up mode, that is enetered either in order to transmit data or to retrieve cached data, or by sending a PS-Poll frame. After the last cached packet is received, the device enters the PSP mode again. We found out that the PSP mode affects packet delay as well as packet delay variation. This is of particular importance in case of data being sent by ORTE at higher frequencies. The Android allows a program to switch to Continuously Aware Mode (CAM) by acquiring `WifiLock` in the `WIFI_MODE_FULL_HIGH_PERF` mode, that is available since Android 3.1.x. After an application acquires this lock the Wi-Fi module will be kept in CAM even when the device enters sleep mode.

We measured the ping responses with the `WifiLock` both acquired and released. We used 1000 packets, each sent at the interval of 200 milliseconds. The results can be seen in Table 1.

Furthermore the PSP mode could cause issues with some Wi-Fi Access Points, that do not support it. According to [12] this could lead to Wi-Fi connection being lost.

WifiLock taken	**Yes**	**No**
Packet loss	0.9%	0.2%
Min. RTT [ms]	1.2	1.9
Avg. RTT [ms]	3.3	59.6
Max. RTT [ms]	177.2	351.7

Table 1: Comparison Wi-Fi ping latencies with and without `WifiLock`.

Another way Android conserves power is that, by default, the incoming multicast traffic is ignored. In order to receive it, the program has to acquire the `MulticastLock`.

Both mentioned locks require specific Android permission to be granted to an application using them. Specifically the `WifiLock` requires the WAKE_LOCK permission and the `MulticastLock` requires the CHANGE_WIFI_MULTICAST_STATE permission.

4 Evaluation

We evaluated our implementation by writing a simple application (Section 4.1) and we also compared the performance of the native and Java implementations (Section 4.2).

4.1 Example Android Application

In order to demonstrate functionality of the ported library an application for remote control of a small robot has been written in Java. The example application is capable of displaying information received from an on-board Laser Range Finder, monitoring battery voltage and controlling robot's motion.

The motion is controlled by an accelerometer embedded in the Android device. To suppress the noise, accelerometer output is filtered by a low-pass filter. We used standard sensor API of Android in order to interface with the accelerometer.

Figure 5 shows a screenshot of the developed application. Figures 3 and 4 then show code of a simple publisher and subscriber. First an application creates a `DomainApp` object, then registers the data type that it uses and finally registers a publisher or subscriber. Publisher can publish new data by calling `send` method, subscribers receive data in the `callback` method.

```
public class ExamplePublisher {

 public static void main(String[] args) {
  NtpTime persistence = new NtpTime(3);
  int strength = 100;
  DomainApp appDomain = new DomainApp(
    0, DomainProp.defaultPropsCreate(),
    null, false);
  ExampleData datamsg = new ExampleData(
    appDomain, "example_topic");
  PublProp publProp = new PublProp(
    datamsg.getTopic(),
    "example_type", persistence, strength);
  Publication pub = appDomain.
    createPublication(publProp, datamsg);

  pub.send(datamsg);
 }
}
```

Figure 3: Code of a simple Java publisher

```
public class ExampleSubscriber
    extends SubscriptionCallback {

 public static void main(String[] args) {
  NtpTime deadline = new NtpTime(10);
  NtpTime minSeparation = new NtpTime(0);
  DomainApp appDomain = new DomainApp(
    0, DomainProp.defaultPropsCreate(),
    null, false);
  ExampleData datamsg = new ExampleData(
    appDomain, "example_topic");
  SubsProp subProps = new SubsProp(
    datamsg.getTopic(), "example_type",
    minSeparation, deadline,
    ORTEConstant.IMMEDIATE,
    ORTEConstant.BEST_EFFORTS, 0);
  Subscription sub = appDomain.
    createSubscription(subProps, datamsg, this);
 }

 public void callback(RecvInfo info,
                      MessageData msg) {
    // do something with the data ...
 }
}
```

Figure 4: Code of a simple Java subscriber

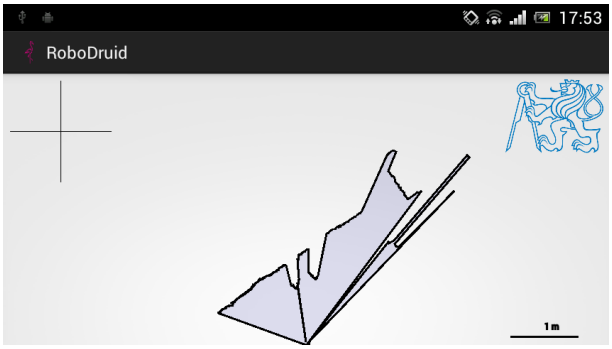

Figure 5: Screenshot of the robot control application. Data measured by the Laser range finder are at the bottom, the speed vector of the robot is at top left.

4.2 Performance Comparison

To have a basic understanding about the overhead of the Java wrapper used for writing Android applications, we conducted a set of experiments. We created a publisher and a subscriber as applications in both C (native) and in Java. The publisher tries to publish data as fast as possible. Because both the publisher and the subscriber were configured as "reliable", ORTE ensures that no publication gets lost and the subscriber receives all the published data. Both the publisher and the subscriber were run on the same device to measure the performance of the middleware itself rather than the performance of the underlying network. We measured how long does it take to publish ten thousands integer values. The experiments were run on three different devices Sony Ericsson Xperia Ray with Android 4.0.4, Google Nexus 7 with Android 4.3 and a PC witch Intel Core i7-3520M CPU running at 2.90 GHz with OpenJDK 7. The results can be seen in Table 2.

Pub → Sub	Xperia	Nexus 7	PC
C → C	2.2 s	2.3 s	0.31 s
Java → Java	10.3 s	6.8 s	0.78 s
C → Java	10.1 s	6.3 s	0.78 s
Java → C	2.6 s	2.5 s	0.31 s

Table 2: Performance comparison of native code and Java wrapper. Time needed for publication of 10000 integer values.

The relative slowdown of the Java implementation compared to the native one is depicted in Figure 6. It can be seen that Dalvik VM performance has improved between version 4.0.4 and 4.3 (or that it runs better on Nexus 7 hardware). OpenJDK performs even better than Dalvik VM. There is no slowdown in Java → C case and in both cases with the

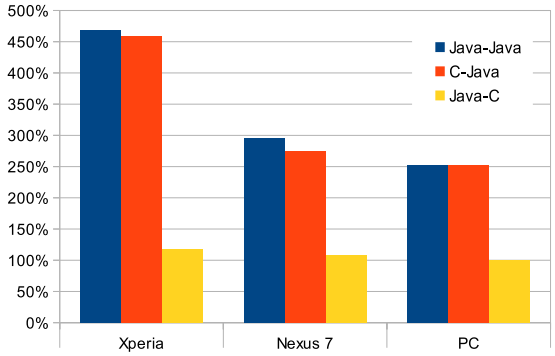

Figure 6: Relative slowdown of Java implementations compared to native (C → C) benchmark.

subscriber in Java, the performance is the same.

The reason for Java subscriber being slow is mostly because of the way Java receive callback is called from the native code. There is a native receive thread, that handles received packets as they arrive. If new data were received, a native callback function is called from within this thread. The callback is passed a buffer containing the received data as well as an instance of `ORTERecvInfo` structure containing information about the data (e.g. endianess, topic, sequence number). In case of Java subscribers this callback is implemented as a JNI function that first attaches itself to the Java VM to have access to Java classes and existing objects. It then creates a new Java object, that is essentially a Java version of the `ORTERecvInfo` structure. After this, it sets byte order of Java's direct `ByteBuffer` according to `ORTERecvInfo` and calls a method of an Java object, which represents the received data, to deserialize the data stored in the `ByteBuffer`. When this is done an application callback function is called with the deserialized Java object as a parameter. After the Java callback returns the native receive thread detaches itself from the Java VM.

As could be seen from the above description the main bottleneck lays in the fact that the native receive thread, from which the Java wrapper's receive callback function is called, has to be attached to the Java VM and has to call many JNI functions that copy a lot of data internally. As could be seen from our measurements the drop in throughput is significant.

A better way to call Java's receive callback function would be to create a dedicated Java thread reading from a custom message queue to which messages could be written directly from the native receive callback function without the need to attach to the Java VM. This is considered for future development.

5 Compatibility with the Latest RTPS standard

ORTE was developed according to the RTPS draft document [6]. For this reason, ORTE is compliant neither with the first adopted RTPS standard [13] nor with its latest revision [4]. In this section we try to summarize the changes that need to be done in order for ORTE to be compliant with the standardized versions. This information might be useful for future ORTE development.

Data with key. RTPS 1.2 introduced a new type of data objects – data with key. This allows to distribute a set of data instances (as opposed to a single data instance) under a single topic. A part of the data instance, called a key, is used to distinguish between different instances.

Discovery protocol. RTPS draft uses a special application called *Manager* that runs on every node and manages automatic discovery of applications both on the same node and on remote nodes. Newer RTPS version replaces this with the *Simple Participant Discovery Protocol* and the *Simple Endpoint Discovery Protocol* that do not need the manager. These protocols use *data with key*.

Data fragmentation. RTPS 1.2 allows big data instances to be fragmented and sent as multiple messages. Receivers re-assemble the data from the fragments. A new "submessage" type `DataFrag` is defined for that purpose.

6 Conclusion

We have successfully ported a Real-Time Publish-Subscribe middleware called ORTE to the Android platform. Now, it is possible to easily write Android applications that communicate over RTPS protocol. The performance of Java publishers is comparable to the native ones, but the performance of Java subscribers is worse due to a bad design decision. This, however, does not limit practical usability, because Android devices like mobile phones usually communicate over a wireless network, which represent the bottleneck.

As for our future work, we plan to make ORTE compliant with the adopted RTPS standard [13].

References

[1] Object Management Group, *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 1: CORBA Interfaces*. [Online], 2008, no. formal/2008-01-04.

[2] ——, "Data distribution service for real-time systems, version 1.2," Online: http://www.omg.org/cgi-bin/doc?formal/07-01-01.pdf, Jan 2007.

[3] M. Sojka, "ORTE web site," Online: http://orte.sf.net/.

[4] Object Management Group, "The real-time publish-subscribe wire protocol, DDS interoperability wire protocol specification (DDS-RTPS)," Online: http://www.omg.org/spec/DDS-RTPS/2.1, Nov 2010.

[5] OpenDDS, "OpenDDS web site," Online: http://www.opendds.org/.

[6] Real-Time Innovations, Inc., "RTPS wire protocol specification, version 1.0," Online: http://orte.sf.net/rtps1.2.pdf, Feb 2002, Draft Document Version: 1.17.

[7] P. Smolík and P. Píša, "ORTE: The Open Real-Time Ethernet," Czech Technical University in Prague, Tech. Rep., 2008, Online: http://orte.sf.net/rtn08_orte.pdf.

[8] K. Tran Duy, M. Žídek, J. Benda, J. Kubias, and M. Sojka, "Autonomous Robot Running Linux for the Eurobot 2007 Competition," in *Ninth Real-Time Linux Workshop*. Real-Time Linux Foundation, 2007, pp. 9–13.

[9] PrismTech Ltd., "Opensplice mobile brings dds to android mobile devices," Online: http://www.prismtech.com/opensplice/products/opensplice-cloud/opensplice-mobile, visited 10/2013.

[10] Google Inc., "JNI tips," Online: http://developer.android.com/training/articles/perf-jni.html, visited 10/2013.

[11] Laird Technologies, Inc., "Power save polling," Online: http://www.summitdata.com/Documents/Glossary/knowledge_center_p.html#psp, visited 10/2013.

[12] Intel Corporation, "Power save polling (PSP) causes connection issues with access points," Online: http://www.intel.com/support/wireless/wlan/sb/cs-006205.htm, visited 10/2013.

[13] Object Management Group, "The real-time publish-subscribe wire protocol, DDS interoperability wire protocol specification," Aug 2006, ptc/06-08-02.