Dynamic Data Distribution: Scalability and Performance

MAREK MAŁOWIDZKI, MICHAŁ MAZUR, PRZEMYSŁAW CABAN
Military Communication Institute, ul. Warszawska 22A, 05-130 Zegrze, Poland

Abstract: In the paper, we discuss timely data distribution in a highly dynamic environment, with a single data source and a potentially very large number of data consumers. We compare two basic data delivery modes: a simple request-response mode (polling) and publish-subscribe, based on the OMG Data Distribution standard. Our case study involves an application of both modes in the Insigma project for timely and efficient delivery of road traffic data.
Keywords: publish-subscribe, request-response, the OMG Data Distribution Service, Java Remote Method Invocation, Web Services

1. Introduction

Timely and reliable data distribution in a dynamic system is a typical, although nontrivial task. In the case of the same, asynchronously generated data to be delivered to multiple recipients, a sound approach is to employ a publish-subscribe approach, dispatching data when they become available and avoiding unnecessary activities (such as periodic checking whether new data are available, as it is in a request-response mode) as well as related processing and communication overhead.

There are a number of mature technologies backed by COTS and open-source implementations, including the Java Message Service, the CORBA-based OMG Notification Service, and a number of proprietary solutions. A recent standard from OMG, the Data Distribution Service [1], is a relatively new player in the field, promising to provide performance and scalability beyond the capabilities of older solutions, and quickly gaining popularity and market acceptance. We thus selected the DDS as a candidate solution to data distribution needs in our road traffic optimization and control system, developed as a part of the Insigma project (described further in the paper). We test a commercial DDS implementation and compare it with a simple polling scheme, focusing mainly on scalability and performance, but also covering other issues (such as fault tolerance and applications portability).

---

1 The work has been co-financed by the European Regional Development Fund under the Innovative Economy Operational Program, INSIGMA project no. POIG.01.02-00-062/09.
The paper is organized as follows: First, some introduction on the Data Distribution Service is made. Then, we briefly describe the Insignia project, focusing on its road traffic subsystem. Further in the paper, we discuss our testing scenario and testbed. Finally, we comment on results of our tests and review related work.

2. The OMG Data Distribution Service

The Data Distribution Service (DDS) [1] is a middleware standard created by the Object Management Group (OMG) for integrating real-time systems. DDS promotes loose coupling between system components. It enables data distribution between many sources and many destinations using the publish-subscribe paradigm.

The DDS introduces a virtual Global Data Space, which allows applications to communicate with each other by reading and writing data objects. Data providers publish typed data, defined by a topic that consumers can subscribe to. The DDS enables an extensive control of QoS parameters such as priority, reliability, delivery deadlines, etc.

The DDS is based on the peer-to-peer (P2P) architecture. Data flow in the DDS is illustrated in Fig. 1. A data publisher first creates a topic, which is an aggregation of a structured data type, a key list, and a specific QoS contract. Then, it may instantiate a data writer, used to actually publish data. On the receive side, a domain participant uses the topic to create a subscriber and then employs a data reader for data reception from other domain participants. The DDS is responsible for handling failures (such as inaccessible data receiver).

The DDS has a number of significant commercial deployments in military, communications, financial and public sectors (listed on vendors’ websites). Recently, a communication protocol has been standardized [2] and interoperability between leading vendors has been demonstrated [3]. For an in-depth discussion on the DDS, refer to [6], [7], and [9].

Figure 1. DDS data flow (DW – Data Writer, DR –Data Reader) (source: [6])
The Data Distribution Service has been standardized by the Object Management Group, the body that has coordinated work on CORBA [4], and, specifically, its Notification Service [5], which has also been designed as a publish-subscribe solution. Additionally, the Notification Service has multiple COTS implementations and many commercial deployments. Therefore, some brief comparison of the DDS and the Notification Service would be appropriate (a detailed comparison is beyond the scope of the paper). It would be quite correct to see the DDS (or, to be more precise, its DCPS layer [1]) as a lightweight version of the Notification Service (or, at least, a simpler tool to handle similar tasks). Firstly, unlike the Notification Service, the DDS is not based on CORBA. Secondly, DDS employs peer-to-peer relationships between data producers and consumers, while the Notification Service is a broker which producers and consumers are connected through (strictly speaking, a concept of event channel is used). Thirdly, in the DDS, published events are type-safe, while the Notification Service employs the Any type and requires appropriate conversions (which is both computation intensive and error prone). There are also a number of other differences, e.g., the quality-of-service issues.

3. The INSIGMA Project

The Inigma project is aimed at developing intelligent information system for global monitoring, detection and identification of threats. The system collects data from various kinds of sensors, cameras, and users, and processes the data to identify threats and notify appropriate public services. One of Inigma’s tasks is road traffic optimization and control.

![Figure 2. The traffic subsystem of Inigma](image)

The key components of the road traffic subsystem (shown in Fig. 2) include the dynamic map server, containing the current data about traffic and associated
events (such as traffic accidents, traffic jams, road works, weather events, etc.). The data are collected from various sources – sensors, the police, public road maintenance services, weather forecast systems, etc. The data should be timely delivered to clients (either public services or plain users) to notify them about the current road situation (and planned changes in traffic organization); the data are also a direct input to the route planning service (which may be an external service or may be available locally, in a “rich” terminal). The problem of data distribution is untypical because of a very large number of clients (terminals), their dynamic nature (continuous movements), and related scalability and performance challenges.

As the DDS standard and its implementations claim to “deliver the performance and scalability,” we decided to consider the service as a data delivery middleware, testing performance of a commercial solution (OpenSplice [16]) and a simple request-response mode against each other. The following sections lay out the details.

4. Testing Scenario

![Figure 3. The testing scenario](image)

Our testing scenario assumes that the whole geographical area of interest (say, the territory of Poland) is divided into a number of tiles (squares). A traveling client is interested in the tile it is currently visiting and its neighboring tiles. Thus, a client needs up-to-date information about a larger area covered by nine tiles (see Fig. 3). The tile “margin” (composed of neighboring tiles) enables a smooth transition of a traveling client from one tile to another. Such a transition requires a change in the area of interest, which involves a number of actions that may take
a noticeable time. In a situation when, for example, a client needs to learn about the current situation in new neighboring tiles, and the publish-subscribe mode is in use, it should additionally unsubscribe from receiving data about some old margin tiles and subscribe for new margin tiles. In practice, to avoid the effect of “vibrations” when crossing back and forth a single tile edge, some margin (delay or hysteresis) should be employed.

The clients gather information about the area of interest using one of the two modes. With request-response, they simply periodically poll the map server, asking about changes concerning the nine tiles since the last query. With publish-subscribe, they subscribe to all nine tiles, waiting for notifications (as mentioned above, a traveling client needs to update its subscriptions accordingly). In our implementation, each tile is represented by a separate topic. This implies a single data type used for describing dynamic events. In addition, we assume that filtering, if required (e.g., by special clients), would only be limited to event type. In other words, to enable maximum performance, the architecture should be as simple as possible.

5. Testbed & Tests

We performed the tests in an isolated (with no additional traffic) 100 Mbps LAN environment shown in Fig. 4. The server and four clients were connected to separate networks through a router to make sure that the DDS service did not use the broadcast mode (we intended to simulate a WAN network in this aspect). Network failures were simulated through disconnecting the link between two Ethernet switches.

Figure 4. The LAN testbed
We developed dedicated software, implemented in the Java programming language. The software architecture is presented in Fig. 5. Event generator was responsible for event stream generation – the first test employed a randomly generated sequence, saved to a file and later repeated in each subsequent test. Event producer either simply forwarded events (in the DDS case) or collected them and returned in responses to clients’ requests (the request-response case). An (optional) web server acted as a gate to the internal Java Remote Method Invocation (Java RMI) server and enabled additional tests comparing two request-response modes: “pure” RMI and Web Services.

![Figure 5. The software architecture (key software components shaded)](image)

The client application was composed of the following key components: vehicle motion simulator, event consumer, and validator. The simulator computed times when vehicles moved from one tile to another (along the route), and generated corresponding stream of topic (un)registrations. The event consumer was responsible for handling communications and delivering received events to the validator. For example, for a request-response mode, the consumer maintained a list of registered topics (one topic per tile) for each vehicle and scheduled requests asking the server about new events that had occurred within these tiles since the last query (a single request for all tiles a vehicle was registered to). Requests were performed asynchronously, using a thread pool. (In the DDS case, communication issues were handled by the service implementation.) Finally, the validator verified collected events (events correctness, sequence order, missing events, etc.) and computed statistics.

We performed tests for the two event delivery modes: publish-subscribe (DDS) and request-response (RMI and Web Services/RMI). Two vehicle speed values were used: 0 (vehicles did not move) and 25 units per hour (a tile side length was
also one unit but the routes used in experiments instructed vehicles to move along
diagonals; thus, registration changes occurred roughly every 200 s.). Additionally,
for DDS, we checked how it reacts to network failures.

We were trying to make the tests possibly close to the real world conditions.
This concerns both scenario (vehicle routes for a single client, did not overlap
to simulate a number of separate terminals) and values of parameters (see Tab. 1).
However, we admit that the amount of hardware was modest and a more reliable
test should employ a larger number of workstations and servers.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event generation frequency (originally generated using the exponential</td>
<td>2000 events per sec.</td>
</tr>
<tr>
<td>distribution)</td>
<td></td>
</tr>
<tr>
<td>Number of tiles</td>
<td>$96^2 = 9216$</td>
</tr>
<tr>
<td>Number of vehicles simulated by each client application</td>
<td>256</td>
</tr>
<tr>
<td>Polling interval (request-response)</td>
<td>10 sec.</td>
</tr>
<tr>
<td>Test duration</td>
<td>30 min.</td>
</tr>
</tbody>
</table>

To make the results comparable, each test repeated the same sequence of events
generated in server and the same vehicle routes on the client side – the only differ-
ence was the events distribution method.

During the test, we used the DDS implementation from OpenSplice [16],
version 4.3, Community Edition. All machines were running Windows 7 and
Java 6.

6. Test Results and Conclusions

In the following paragraphs, we discuss the tests results.

**Network traffic volume**: As expected, the DDS service generated low traffic
– except for the initial discovery phase (Fig. 6). Client registrations at maximum
speed caused increased traffic from clients to server. For both RMI and WS, requests
generated more traffic than server responses – as many of them were empty (did
not contain any event). This additional traffic is the cost of a relatively short event
delivery time (roughly equal to half the polling period).
Figure 6. Traffic originating from server (upper row) and clients (bottom row) at speed 0 (left column) and speed max (right column), in MB/s, averaged per 10 s.

Table 2. Network traffic details (times: min/max/avg)

<table>
<thead>
<tr>
<th>Protocol stack</th>
<th>DDS</th>
<th>RMI (JRMI)</th>
<th>SOAP/HTTP 1.1/TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request size (9 topics)</td>
<td>–</td>
<td>–</td>
<td>540 (min) 1220 (max)</td>
</tr>
<tr>
<td>Reply size (0 events)</td>
<td>–</td>
<td>450 (min) 606 (max)</td>
<td></td>
</tr>
<tr>
<td>Reply/message size (1 event)</td>
<td>216 (min) 700 (max)</td>
<td>780 (min) 1032 (max)</td>
<td></td>
</tr>
<tr>
<td>Reply size (2 events)</td>
<td>–</td>
<td>780 (min) 1032 (max)</td>
<td></td>
</tr>
<tr>
<td>Reply size (3 events)</td>
<td>–</td>
<td>862 (min) 1229 (max)</td>
<td></td>
</tr>
<tr>
<td>Control message sizes</td>
<td>200 – 1300 (min) 287 – 451 (max)</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>Control vs. event traffic [pk]</td>
<td>35% (speed 0) – 29% (TCP ACKs excl.)</td>
<td>55% (TCP ACKs excl.)</td>
<td>–</td>
</tr>
<tr>
<td>Control vs. event traffic [B]</td>
<td>31% (speed 0) – 25%</td>
<td>92%</td>
<td>–</td>
</tr>
<tr>
<td>Total server traffic [1000 pk]</td>
<td>690 (speed 0) – 862</td>
<td>680 (min) 2040 (max)</td>
<td></td>
</tr>
<tr>
<td>Total server traffic [MB]</td>
<td>197 (speed 0) – 250</td>
<td>220 (min) 506 (max)</td>
<td></td>
</tr>
<tr>
<td>Server response time</td>
<td>–</td>
<td>&lt;1/519/&lt;2</td>
<td>3/3473/22</td>
</tr>
<tr>
<td>Event delivery delay</td>
<td>few ms/250/50</td>
<td>few ms/10000/5000</td>
<td>500/12000/6000</td>
</tr>
<tr>
<td>Average event delivery jitter</td>
<td>45 (min) 3600 (max)</td>
<td>–</td>
<td>3700</td>
</tr>
</tbody>
</table>
**Network traffic details:** Data concerning protocols, server traffic (round-trip), message sizes (application-level payload sizes in bytes) and delays (in milliseconds) are given in Tab. 2. Standard RMI and Web Services versions were used in test (no custom mechanisms such as serialization or compression were applied). Note that delay times for Web Services account for an additional “hop” – i.e., communication via RMI with internal server, and that this step could be avoided in a real-world implementation.

The DDS traffic pattern heavily depends on how often the topic registrations change – but even when registrations do not change at all (at speed 0), control DDS traffic accounts for almost 70% of the total traffic volume. The overall server traffic results are comparable with RMI; however, in the RMI case, clients poll for new events every 10 s. (which yields an average event reception delay of 5 s.), while DDS delivers events with a minimum delay. RMI requires about 80 bytes per each event, while Web Services requires about 200 bytes. In addition, a web server response requires at least two packets, the first one carrying the HTTP response (strictly speaking: the HTTP header) and subsequent ones containing a web response.

**Network failures (the DDS case):** Unfortunately, even a short network failure caused the service to break; the service was unable to recover after a network repair and no more events were delivered to clients running on previously disconnected machines. The packet trace analysis suggests that a typical DDS communication pattern was re-established after a repair, although events did not reach the event consumer. We are not sure about the reason – the protocol is documented as resistant to communication problems. Due to the “failure” of the test, we were unable to verify whether the service guarantees event delivery even during a network breakdown (i.e., whether it supports event retransmission). However, we think that in our scenario such guarantees should *not* be used, as the requirement for a server to cache events until successful delivery to every client would be practically impossible to fulfill (assuming a *really large* number of clients).

**Scalability:** The request-response mode is scalable by definition (i.e., no per-client state is held in server); the only limitation is the server (or, a server farm) and its ability to handle a high volume of requests. In the case of DDS, the server maintains client registrations and exchanges non-negligible discovery traffic; still, it is also possible to divide the whole area (map) between a farm of cooperating servers and thus mitigate the load on individual servers.

For DDS, our modest hardware setup allowed simulating a relatively small number of vehicles – 1024 (256 per each client machine). Selecting a larger number of tiles (and thus, a larger number of simulated vehicles) caused problems with the implementation – it crashed after a while with a native exception (access violation). Changing the database size in a configuration file helped to some degree, but due to various problems, we did not succeed in performing tests for larger scenarios. Definitely, more tests would be needed to verify scalability (and we be-
lieve that the number of simulated vehicles should reach tens of thousands) but we think an updated (and less troublesome) version of the service should be selected for further experiments.

**Final observations & conclusions:** The main strength of the DDS service is that it is fast and relatively lightweight (in terms of network overhead). However, for simplicity, we omitted a “tile setup” phase – a vehicle entering a new tile must first gather information about the tile’s current state (i.e., about active events). Additionally, in our scenario, a guaranteed event delivery method is required. (Some “heartbeat” mechanism would also be needed to verify that the server is accessible during silence periods – but this could be provided by artificial events, generated periodically by the server for each tile. Another possibility is to receive feedback from the DDS layer.) Thus, the DDS service alone is not sufficient – it must be supplemented with the request-response mode for the tile setup phase and later, for retransmissions of lost events. This complicates software development but does not exclude such a mixed-mode approach. Of course, a preferred protocol is a lightweight, binary one – it seems that there are no clear arguments for employing Web Services here.

Of course, selecting different values of key parameters would affect the results. For example, a shorter polling period (for request-response) would result in a smaller event delivery delay but cause a large traffic overhead. Nonetheless, we believe that the values we have selected are sound (in the scenario being considered) and give a reasonable basis for the comparison.

Perhaps it would be instructive to compare the OpenSplice-specific protocol, used in the tests, with the interoperable standard, the DDS Interoperability Wire Protocol (DDSI) [2]. Intuitively, we think the custom solution should perform better.

Finally, one should note that we do not discuss security – in a real-world application, events would require a digital signature, which would affect performance and perhaps introduce additional issues (e.g., the DDS and request-response modes could require different security mechanisms).

### 7. Related Work

Xiong et al. [8] (see also a summary in [9]) compare three commercial DDS implementations against other publish-subscribe solutions such as the Notification Service, JMS and GSOAP and find that all tested DDS products deliver significantly better performance – and, despite wide differences in performance between these products, they are at least twice as fast as their competitors. The work also contains evaluation of the DDS standard, pointing out its strengths and weaknesses, and a comparison with CORBA (although, in our opinion, the DDS should rather be compared with the Notification Service). Additional results are presented in [10]. Carrozza at al. [15] compare performance of the DDS versus the Java Message Service. Reference [12] reports a significant performance boost of a robotics framework
after it has been based on OpenSplice. Other interesting studies [13], [14] employ mathematical models and simulation, and find the publish-subscribe concept suited well for real-time data delivery.

8. Summary

In the paper, we analyze the application of the Data Distribution Service for reliable and timely data delivery in a highly dynamic environment, with a potentially large number of clients. We verify the performance and scalability of DDS and compare it with a simple request-response, polling-based approach. The main conclusion is that DDS (strictly speaking: the tested implementation) is fast and lightweight, although it must be supplemented with the request-response mode. In addition, our results shed some light on scalability of a commercial DDS implementation, although further experiments, involving tens of thousands of clients, are needed. We believe that the results of our study are universal and may be utilized as a decision factor for other DDS deployments, such as tactical picture sharing (a problem very close to our study) or sensor network surveillance.

REFERENCES