Integrating Third-party Services Using Brokers in the Serious Games' Domain

Stefan Dimitrov Stavrev [1], Todorka Zhivkova Terzieva [1], Angel Atanasov Golev [1]

[1] *Faculty of mathematics and informatics, Plovdiv University "Paisii Hilendarski", 24 Tsar Asen, 4000 Plovdiv, Bulgaria*

***Abstract* – In this paper we demonstrate how to integrate 3rd party services in serious games. We use message queue broker and micro-services in a publish/subscribe manner in order to use real-time 3rd party data into a serious game's logic. First, we discuss the benefits of service oriented architecture. Then, we analyse and compare different message queues brokers in terms of data latency, throughput, fail-tolerance and scalability for the purpose of serious games. As a sequence, we apply those best practices from other domains in the field of Serious Games (SGs). Finally, we summarize the presented ideas and comparisons and draw conclusions.**

***Keywords* – Serious games, Message oriented middleware, systems integration, message queue.**

1. Introduction

Message queues (MQ) have been around since the 80's. They have the benefit of handling and exchanging large amounts of data between different systems. They have already been successfully applied in various domains – handling transactions in bank institutions and stock exchange, handling real-time messages for social networks like Facebook and LinkedIn. Message queues are used in operating systems to route mouse and keyboard input. MQ brokers are linearly scalable, able to work in nodes and even in the cloud [13]. However, their application in the video games industry and in the serious games domain in particular has remained somewhat limited. Multi-player servers still rely on the request/response paradigm. Services and service oriented architecture [2], on the other hand, is an emerging trend in the field of serious games development.

2. Previous work

Recent research in the field of Serious Games is conducted on architecture that is oriented entirely towards services [4,3]. Service-Oriented Architecture (SOA) is a set of practices for architectural design of software that exploits services as loosely coupled components orchestrated to deliver various functionalities. The SOA paradigm is not well established in the SG domain, yet. The components provide independent services to other components of the serious game or application. The key principles in this particular design are modularization and re-use of functionalities. That concept in not new in the field of computer science but is relatively rarely applied, yet, in the field of games and serious games in particular [10,11]. Additional benefit of using services is the lack of compile-time dependencies. Moreover, it is entirely possible to have the core gaming as a service in a centralized server. But the biggest advantage remains the re-use of components [7] - shared user profiles, knowledge databases on learning topics, natural language processing dialog services, exchanging scores between different game instances. Of course, the SOA approach is not without shortcomings. Some of the challenges are that the quality assurance and testing module integration tend to be more difficult when developing SOA applications. In addition, sometimes the lack of documentation on the usage of interfaces makes integration with a certain service difficult. Furthermore, extra attention to services description needs to be kept in mind. Another limitation of the SOA approach is that the game needs to be constantly online, i.e. connected to a certain service or services. That last restriction makes the

DOI: 10.18421/TEM74-23
https://dx.doi.org/10.18421/TEM74-23

Corresponding author: Stefan Dimitrov Stavrev, *Faculty of mathematics and informatics, Plovdiv University "Paisii Hilendarski", Plovdiv, Bulgaria*
Email: stavrev@fmi-plovdiv.org

Received: 28 July 2018.
Accepted: 22 October 2018.
Published: 26 November 2018.

architecture less flexible. Finally, there is the additional performance cost due to network calls.

Some of those shortcomings can be avoided by getting service data in a publish/subscribe manner instead of direct request/reply [13]. The publish/subscribe methodology is a well know programming paradigm in the field of computer science - it allows loose coupling between system components [1]. It has been successfully applied in various domains where complex subsystems, possibly written in different programming languages, need to exchange information [5]. An additional benefit is that the constant internet connection problem in no longer a requirement by using a message queue in between the game and the 3rd party service.

Message queues are middleware systems that enable developers to have fault-tolerant, distributed, decoupled, service oriented architecture. But why not use traditional request/response pattern for serious games? First, let us take a closer look at the problem we are presented with. The typical requirements are that SGs should be light, decoupled, possibly run on low-end hardware, be easily scalable and make use of 3rd party services [10, 11]. Using a traditional request/response pattern (REST, for instance) from inside the game logic will put additional demand on the hardware resources which we want to avoid. The constant requesting whether there is new data available from the 3rd party server is a cumbersome operation. An event-driven server architecture (with web-sockets, for instance) can potentially solve the constant data polling demand but still the game logic will be tightly coupled with the 3rd party services. That in turn sacrifices application scalability. On the other hand, real-time data is highly dynamic by nature. In the event of a network failure, 3rd party data will be lost. Another advantage of using MQs over traditional direct messaging is their asynchronous nature, which allows us to put a message on the queue without processing it immediately.

3. Previous work

Adding a MQ broker in between the game logic and the 3rd party service solves some of the integration problems with SOA. However, there is a number of messaging brokers available; most of them are open-source, others are proprietary. When it comes to choosing the right one we must keep in mind that there is no silver bullet – no one single solution can fit all requirements [16]. That is why we will briefly compare the most widely-used brokers and pick one to be used with our SGs architecture – DiAS [12]. In addition, we must keep in mind the individual broker's design intent. In that context, there are several broker metrics that are important for us:

- Throughput of messages
- Low-latency
- Number of protocols supported out of the box
- Number of features supported out of the box
- Fault-tolerance and data recovery
- Scalability

A very good collection and description of different messaging systems is put on [21]. However, since there are too many brokers available and each of them is created for a different purpose, we are not going to compare them directly. Instead, we will investigate 3 of the most popular, general purpose brokers. The 4-tier architecture that we propose later in this paper allows for changing the broker at any time with little configuration.

3.1. ActiveMQ

The first broker we will investigate is Apache ActiveMQ [22]. It is written in JAVA and is based on JMS [14]. It is open source, under the Apache 2.0 license. Because the broker is JMS-based, it supports 2 types of messaging: topic based and queue based. The queue based messaging is point-to-point: a sender (or also known as publisher) sends a message to a queue and the message is received by exactly one receiver. The other type of messaging is publish/subscribe: a publisher sends a message to a topic and all subscribers to that topic are going to receive it. ActiveMQ supports several wire level protocols: OpenWire, STOMP, MQTT, REST (HTTP GET/POST), AMQP [6]. ActiveMQ supports 3 types of message persistence: AMQ message store (fast read-write), non-journaled JDBC [15] (reliable but not fast) and journaled JDBC (reliable and faster than JDBC). Last but not the least, ActiveMQ offers a nice web-console to monitor and manage the broker traffic in real-time.

3.2. Kafka

The next on the list is Apache Kafka. It is a broker written in SCALA. Kafka is a persistent, distributed, replicated publish/subscribe messaging system [23]. It typically consists of a cluster of brokers. The brokers are stateless, i.e. consumers maintain their own state (with the help of ZooKeeper [27] by default). Kafka has only topics, which can be tuned to act as queues if needed. For message transport, Kafka uses its own binary protocol over TCP. Clients can interface with the broker via web sockets. Apache Kafka has fewer features out of the box – it is built for performance and high throughput. Kafka is created with horizontal scaling in mind. Unfortunately, Apache Kafka ships with built-in support for JAVA clients only.

3.3. RabbitMQ

The last broker to consider in our list is RabbitMQ [25]. It is written in the Erlang programming language. Communication in RabbitMQ can be either synchronous or asynchronous. Publishers send messages to exchanges which are something like group mailboxes. After that, exchanges resend those messages to queues. Finally, consumers retrieve messages from those queues. The use of exchanges which reroute messages to queues is required because RabbitMQ implements the latest specification of the AMQP [6]. With additional plugins however, RabbitMQ has added support for JMS, STOMP and HTTP clients. RabbitMQ has only queues (because of the AMQP again). RabbitMQ uses central node for message routing. That is why it is most suitable for vertical scalability scenarios. In contrast with Kafka, the broker keeps track of consumer state.

4. Messaging protocols

4.1. AMQP

AMQP [6,16], stands for Advanced Message Queuing Protocol. The idea behind its development was to replace the existing proprietary messaging middleware [8, 10]. The main reasons to use AMQP are reliability and interoperability. Out-of-the-box AMQP comes with a variety of messaging features. Some of those features are: topic-based publish-and-subscribe messaging, reliable queuing, security, routing, and transactions. The protocol exchanges the required messages directly by either a topic, or via headers. AMQP is a binary executable, and works on the application level. It is designed to efficiently support a wide variety of messaging applications and communication patterns. AMQP only has queues which in turn are only consumed by a single receiver. AMQP data client applications are not designed to publish messages directly to queues. Instead, a message is published to an exchange, which through its bindings may get sent to one or multiple queues.

4.2. MQTT

MQTT (Message Queue Telemetry Transport) [16, 24] was originally developed by one of the teams in IBM. MQTT was designed to provide publish-and-subscribe messaging (no queues). It was also envisioned to be used with resource-constrained devices and low bandwidth scenarios, high latency networks such as dial up lines and satellite links, for instance. It can be used effectively in embedded systems. MQTT has the advantage of being low footprint and it makes it ideal for today's "Internet of Things" style applications. MQTT offers three qualities of service (QoS):

- At most once / unreliable - QoS level 0
- At least once, to ensure it is sent a minimum of one time (but might be sent more than one time), QoS level 1
- Exactly once, QoS level 2

MQTT's strengths are simplicity, a compact binary packet footprint, and it makes a good fit for simple push messaging scenarios such as humidity updates, wind speeds, oil pressure feeds, stock price movements or mobile notifications [16].

4.3. STOMP

STOMP (Simple/Streaming Text Oriented Messaging Protocol) is text-based protocol and is very similar to HTTP [16, 26]. STOMP provides a message (also known as "frame") header with properties, and a frame body, similarly to AMQP. The intent behind STOMP was to create simple, yet widely-interoperable protocol. For instance, one can use a telnet client to connect to a STOMP broker. By design, STOMP does not deal in queues and topics. Instead, it uses a SEND semantic with a "destination" string [16]. The broker must map onto something that it understands internally such as an exchange, queue or topic. Consumers then SUBSCRIBE to those destinations. Since those destinations are not mandated in the specification, different brokers may support different implementations of the term "destination". That is why, it usually takes an effort to port code between brokers. However, STOMP is simple and lightweight, with a wide range of language bindings.

To summarize, AMQP supports only queues, with the possible emulation of topics. It is supported in: Apache ActiveMQ, RabbitMQ, ApacheQpid. MQTT is de facto the IoT data exchange protocol. It has topics but no queues. It is with light footprint, supported by various brokers: ActiveMQ, moquette, WebSphereMQ. STOMP is the middle grounds between the two protocols. It is lightweight with a number of commands. It is supported by ActiveMQ, RabbitMQ, Gozirra, Sprinkle and many others.

Having reviewed different MQs and the wire level protocols at which each of them supports, we have chosen Apache ActiveMQ (with STOMP) to use for 3rd party integration with serious games. The reason is partially due to the wire level protocols that each broker supports. As we have seen, Active MQ has the largest client protocol support. In contrast, RabbitMQ only comes up with AMQP by default (which ActiveMQ also supports). Kafka, on the other hand, implements its own binary protocol and while it is fast it is also not very generic.

5. Services integration

First, let's revise our distributed gaming architecture – DiAS [12]: *Figure 1*. In DiAS, we presented how a game instance is able to run independently of its input layer.

In this paper, we will pay closer attention to the integration of 3rd party services that may run on the cloud. That part of the architecture is marked in red in Figure 1.

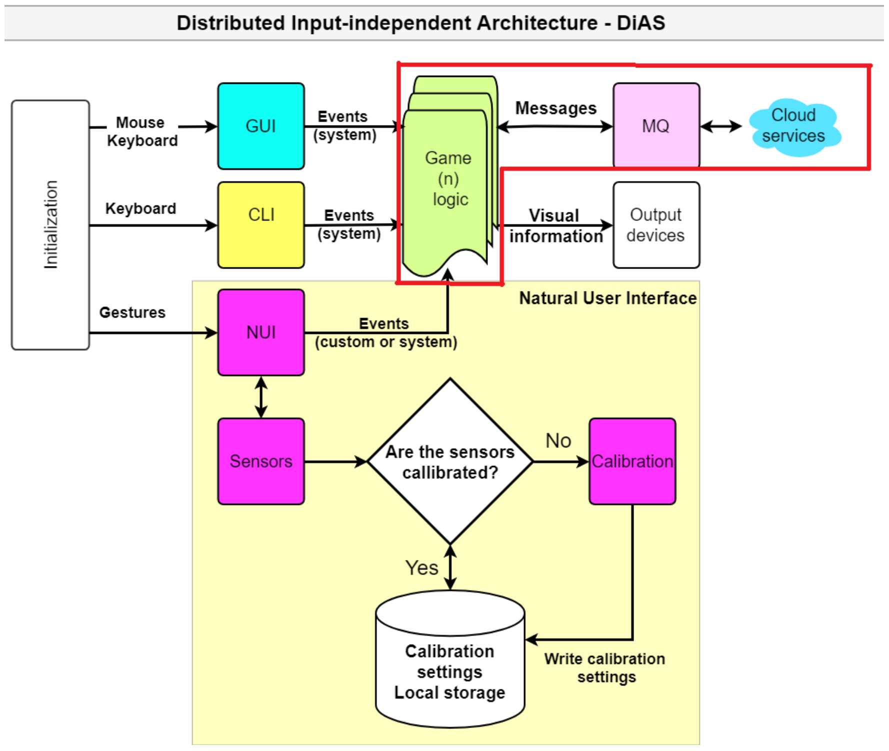

Figure 1: DiAS serious game architecture.

Why do we need 3rd party services data into our SG in the first place? First, because the trend nowadays is oriented towards real-time services, especially SGs [2, 3, 9]. Secondly, we can better capture and reflect the real-world conditions into the DiAS architecture. Imagine the following use case: a player enters the digital environment. Instead of a fixed day-night cycle and weather conditions, that information can be provided in real-time via a 3rd party weather service (Figure 2.). Another example is: we want a school class schedule to be available and visualized inside our game. Serious games usually keep track of scores – how much points have each individual player accumulated during the course of the game. What if we want to compare that score to the scores of all students across the whole school – some mechanism needs to be put into place that collects, aggregates, arranges and finally – presents the summarized scoring information to a certain player. How do we integrate such a scoring system across different classes or even – different schools?

Let's take a look at Figure 2. We want to integrate different 3rd party data sources into our game in order to make it more dynamic and the gameplay – service-oriented. To achieve that, we define a 4-tier integration architecture, consisting of: data source connectors, broker, clients and application.

5.1. Connectors

The first step is to connect to the 3rd party service the provider API, get the data that interests us and later – process it. There are several ways those tasks can be achieved. We can, of course, program our custom logic layer for each individual data source. That approach, however, will have a bigger overhead for adding new data sources in the future. Another possibility is to use Enterprise Integration Patterns [1]. Luckily enough, there are several implementations of those patterns – Apache Camel [17], Microsoft BizTalk Server [18] and IBM WebSphere Application Server [19].

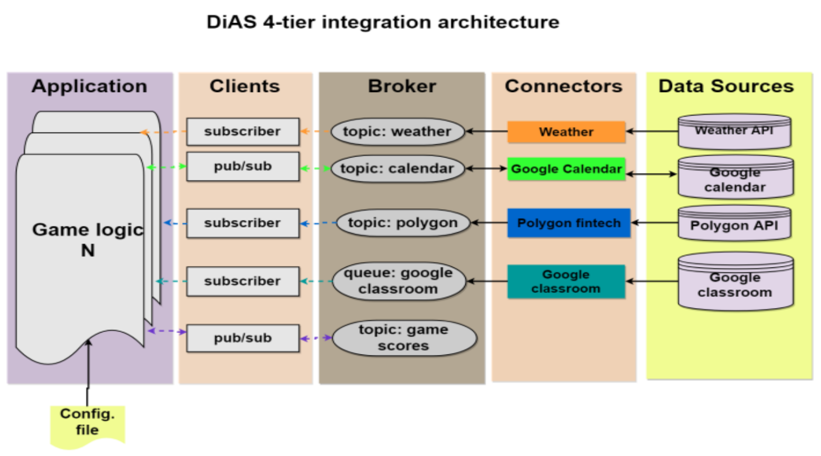

Figure 2: DiAS 3rd party integration architecture

Without going into deep comparison between the mentioned approaches, we chose Apache Camel as our integration framework. The main reason for that decision is:

- Camel is open source
- It supports ActiveMQ out of the box
- Supports JMS
- Is highly configurable with different data sources connectors and processors
- Easy to add message endpoints.

Please note that 3rd party data comes available in different format – XML, JSON, CSV, etc. In addition, different services support various communication protocols: SOAP, REST, web sockets, MQ topic/queue. Apache Camel comes with a built-in support for the most common protocols and data formats. If a certain data format or communication protocol is not supported, it can easily be added. Another integration pattern that we use is the notion of a data processor. It applies data transformations from one data type to another. If the source data needs to be cleaned and filtered, that is a task for the data processor. Finally, the processed data is published to an endpoint. In our case, that endpoint is the MQ broker.

5.2. Broker

The second tier of our DiAS integration architecture is the MQ broker. We have discussed several MQ brokers and chosen Apache ActiveMQ. It is lighter than Kafka, has built in support for various communication protocols – AMQP, MQTT, STOMP, Open wire. In addition, ActiveMQ has direct support for Camel Endpoints [22]. JMS (on which ActiveMQ is based) has the notion of topics and queues. Topics and queues inside a JMS broker are different delivery mechanisms. The main difference between the two is the way messages are delivered and the number of receiving entities. A topic is a one-to-many delivery mechanism. It is the classic implementation of a publish/subscribe paradigm. Messages are delivered to a certain topic by the publisher. Interested entities subscribe to receive updates on certain topics. If there are zero subscribers to a certain topic, the data will still be available on the topic but will expire after a period of time. There is no mechanism to notify the publisher that one of the subscribers has received a message. Queues, on the other hand, are point-to-point delivery mechanisms. They are an implementation of the push / pull mechanism. A publisher, in this case, will put data to a queue. If there is a subscriber to that queue, the message will be received by the subscriber. If there are multiple subscribers to that queue, the data will be received by exactly one subscriber. If no subscribers are available at the time of the data delivery to a queue, the data will be persisted until there is one. The receiving entity has to notify the sender that it has received the message – in contrast, in the topic model, the subscribers have no such obligation. In the event that multiple messages are delivered one-after-the-other to a queue, once a subscriber becomes available, it will receive all the messages for that queue in a FIFO manner. Another aspect that ActiveMQ introduces is the notion of durability [20]. A durable topic subscriber will receive all the message send to that topic, even if that subscriber is disconnected for a period of time. In contrast, with non-durability, the subscriber will receive only the messages sent during an active subscription session.

For integration with DiAS, we propose the following distinction: for 3rd party services, that need to be available to a number of game instances, to use topics. For one-to-one services, such as Google Classroom, to use queues, one per each game instance or classroom. For game scores exchange, to use durable topics. The benefit of the latter is that even in the event of a network spike, peers will still be able to publish and subscribe to game data locally – via the internal network. That statement is valid in the event that the MQ broker resides internally for the local network (for instance, for a class C private network, on tcp://192.168.0.10:61613).

5.3. Client

The 3rd tier of the proposed integration architecture is that of the clients. The clients can be pure subscribers, pure publishers or hybrid – both subscribers and publishers of data. A client can subscribe to a broker topic or queue, depending on the use case. For instance, it makes sense that a weather service client is only a subscriber for weather-related topic since weather data is processed in only one direction. On the other hand, a client for Google Calendar will play a role of both a subscriber and a publisher. It can get data from the public calendar for a particular class (as a subscriber) on a queue and publish data to Google calendar. Another example for a hybrid client is the one responsible for getting and publishing local game scores to and from the broker. That way, different players that play their own instance of the same serious game can view in real-time how well they performed compared to their peers.

5.4. Application

The fourth and final tier is the application. It is here that our game logic resides. One can make the point that the 3rd and 4th tiers can be combined into one. While that is true if we choose that approach, we lose decoupling of the game logic from 3rd party services. In addition, we add a dependency in our game to a certain protocol and broker.

6. Conclusion

In this paper we presented how 3rd party services can be made available in the context of serious games. We were able to achieve that by decoupling the game logic from directly querying for the 3rd party data and introducing a broker to handle the message load. Moreover, we showed a 4-tier integration architecture for delivering 3rd party data, consisting of: data source connectors, broker(s), clients and application(s). It is important to note that each of the 4 tiers is generic and can be substituted, depending on the use cases and the specific needs that a serious game is trying to achieve.

Acknowledgements

Parts of this work are supported by a project SP17-FMI-011 of the Scientific Research Fund of Plovdiv University "Paisii Hilendarski", Bulgaria.

References

[1]. Hohpe, G., & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

[2]. Carvalho, M. B. (2017). *Serious games for learning: a model and a reference architecture for efficient game development*(Doctoral dissertation, Technische Universiteit Eindhoven).

[3]. Carvalho, M. B., Bellotti, F., Hu, J., Hauge, J. B., Berta, R., De Gloria, A., & Rauterberg, M. (2015, July). Towards a Service-Oriented Architecture framework for educational serious games. In *Advanced Learning Technologies (ICALT), 2015 IEEE 15th International Conference on* (pp. 147-151). IEEE.

[4]. Carvalho, M. B., Bellotti, F., Berta, R., De Gloria, A., Gazzarata, G., Hu, J., & Kickmeier-Rust, M. (2015). A case study on service-oriented architecture for serious games. *Entertainment Computing*, *6*, 1-10.

[5]. Black, U. (1995). TCP/IP & Related Protocols. McGraw-Hill, 122-126.

[6]. Vinoski, S. (2006). Advanced message queuing protocol. *IEEE Internet Computing*, *10*(6).

[7]. Van der Vegt, W., Westera, W., Nyamsuren, E., Georgiev, A., & Ortiz, I. M. (2016). RAGE architecture for reusable serious gaming technology components. *International Journal of Computer Games Technology*, *2016*, 3.

[8]. Mahmood, S., Lai, R., & Kim, Y. S. (2007). Survey of component-based software development. *IET software*, *1*(2), 57-66.

[9]. Söbke, H., & Streicher, A. (2016). Serious Games Architectures and Engines. In *Entertainment Computing and Serious Games* (pp. 148-173). Springer, Cham.

[10]. Aalst, W. M. P. v. d., Beisiegel, M., Hee, K. M. v., Konig, D., & Stahl, C. (2007). A soa-based architecture framework. *International Journal of Business Process Integration and Management*. *2*(2), 91–101.

[11]. Sprott, D., & Wilkes, L. (2004). Understanding service-oriented architecture. *The Architecture Journal*, *1*(1), 10-17.

[12]. Stavrev, S., Terzieva, T., & Golev, A. (2018). Concepts for distributed input independent architecture for serious games. CBU International Conference Proceedings 2018: Innovations in Science and Education, Prague, Czech Republic. (in press).

[13]. Skeen, D. Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview. Retrieved from: http://www.vitria.com/

[14]. JMS, Java Message Service. Retrieved from: https://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

[15]. JDBC, Java Database Connectivity. Retrieved from: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

[16]. VmWare blog page. Retrieved from: https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html

[17]. Apache Camel. Retrieved from: http://camel.apache.org/enterprise-integration-patterns.html

[18]. Microsoft Biz Talk Server. Retrieved from: https://www.microsoft.com/en-us/cloud-platform/biztalk

[19]. IBM WebSphere Application Server. Retrieved from: https://www.ibm.com/cloud/websphere-application-platform

[20]. ActiveMQ. Retrieved from: http://activemq.apache.org/how-do-durable-queues-and-topics-work.html

[21]. A list of widely used message queue brokers. Retrieved from http://queues.io/

[22]. Apache Active MQ. Retrieved April 10, 2018 from http://activemq.apache.org/

[23]. Apache kafka. Retrieved from: https://kafka.apache.org/

[24]. MQTT, Retrieved from: http://mqtt.org/

[25]. Rabbit MQ. Retrieved from: https://www.rabbitmq.com/

[26]. STOMP. Retrieved from: https://stomp.github.io/

[27]. ZooKeeper. Retrieved from: https://zookeeper.apache.org/ .