Investigating different Instrumentation Techniques in the context of ESB Runtime Verification

Gabriel Dimech[1], Christian Colombo[1], and Adrian Francalanza[1]

University of Malta

With the increasing popularity of the service-oriented architecture, enterprise software is increasingly being organised as a number of services interacting over an Enterprise Service Bus (ESB). The advantages of this arrangement are numerous, not least the modularity and flexibility it affords and the ease with which multiple technologies can be integrated [2].

Yet, the highly dynamic nature of ESBs — where components can be introduced, replaced, or withdrawn transparently — brings with it a number of challenges, particularly to ensure the correct overall behaviour of the ESB [4]. In a flight booking ESB application, such as that depicted in figure 1, back-end airline services may be swapped at runtime, potentially causing unexpected behaviour. The reason for this may be that during the testing phase not all banking services may have been tested thoroughly or that the service experiences an update without the flight booking application being notified.

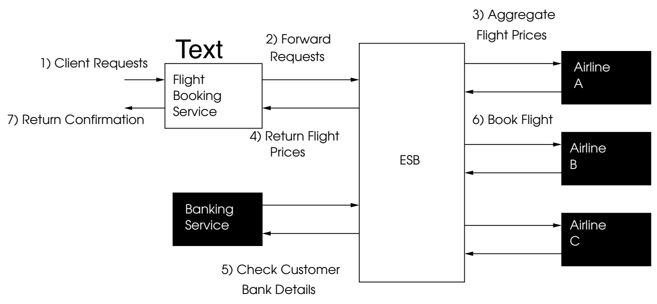


Fig. 1: Flight Booking ESB Application

One way of ensuring that the ESB performs as expected is to check its behaviour at runtime rather than attempting to preempt the possible scenarios the ESB might find itself operating in. While this approach, known as Runtime Verification (RV), introduces an overhead, it ensures that anything which occurs at runtime adheres to a correctness specification [3]. In the context ESB applications, one main concern is to ensure that each incoming message is routed correctly [4]. For instance in the flight booking example, each request must be successfully routed to the correct airline and banking services before an acknowledgement is sent to the customer.

The correctness specification in the context of RV is typically specified in a mathematical notation and the runtime verifier is synthesised automatically. This introduces more confidence that the verifier is itself correct. However, other than correctness concerns, RV also raises overheads concerns which might have a negative impact on the performance of the ESB [1].

Of particular concern in this regard are the overheads related to intercepting relevant events which the runtime verifier would consider for correctness. This is because while the checking itself can be delegated to other processing resources, detecting and transmitting events to the verifier cannot. Through this work we attempt to investigate different options which can be considered for ESB events gathering by comparing the resulting overheads for each approach at runtime.

We start by identifying the different points at which relevant ESB events can be intercepted. In particular, we identify Mule ESB on which to perform our research, mainly due to its popularity amongst enterprises. Mule is a Java based open source ESB supporting XML specifications of Flows which may be used to configure SOA-based applications [5]. This arrangement exposes three different levels of event interception:

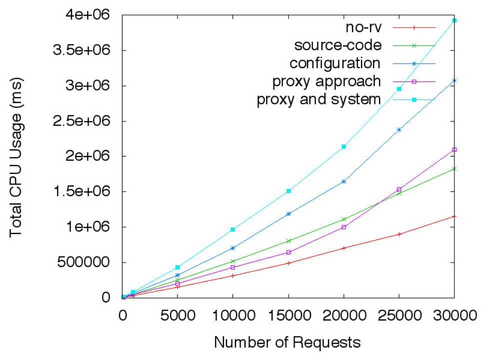
– One may intercept events at the source code level using techniques such as Aspect-Oriented Programming (AOP). This is by far the most commonly used technique for runtime verification of Java systems [6].
– At a higher level, one may include the verifier in the XML flow specification and divert copies of the messages to the verifier component for scrutiny.
– Finally, at the opposite end of the spectrum, the ESB depends on lower-level communication protocols such as HTTP, JMS, FTP etc. to connect components [5]. Thus, one could intercept such communication through a proxy and the verifier is able to intercept events for checking.

Each of these modes of events gathering provides its advantages: Intercepting events at the source code level provides full visibility of the ESB, including internal actions which might not otherwise be visible through the other modes. On the other hand, intercepting messages at the Flow specification level is an approach which allows the user to specify correctness properties more easily. Finally, the proxy approach decouples monitoring code from the ESB, making it easier to dynamically change which information is intercepted whilst reducing direct impact on the ESBs resources.

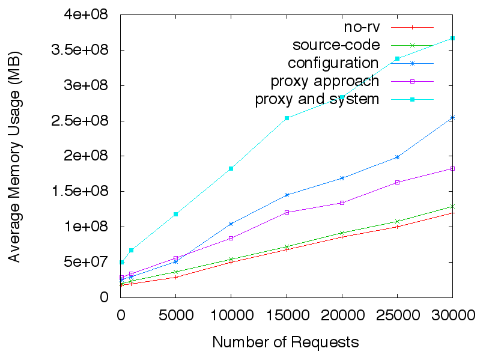
Notwithstanding these aspects, in this study we choose to focus on the performance issues of each approach. More precisely we consider two aspects of performance: the resources required to support runtime verification and the impact of this resource take-up on the end user experience.

Resource Consumption In this regard, we consider the CPU and memory used for each of the approaches.

User Experience To estimate the impact on user experience we measured the latency, i.e. the duration of time needed for a request to get a response, and the throughput, i.e. how many requests can be handled for a particular period of time.

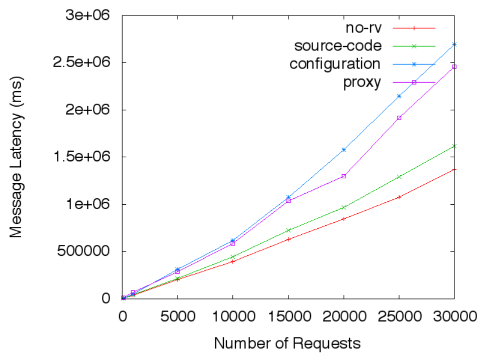


(a) CPU Usage

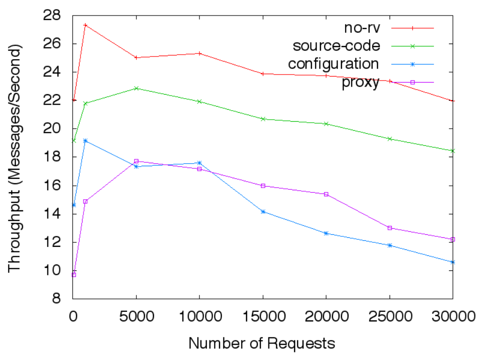


(b) Memory Usage

Fig. 2: Resource Consumption Metrics



(a) Message Latency



(b) Message Throughput

Fig. 3: User Experience Metrics

References

1. E. Bodden. Efficient and Expressive Runtime Verification for Java. In *Grand Finals of the ACM Student Research Competition 2005*, 03 2005.
2. D. A. Chappell. *Enterprise Service Bus: Theory in Practice*. O'Reilly, 2004.
3. M. Leucker and C. Schallhart. A Brief Account of Runtime Verification. *JLAP*, 78(5):293–303, 2009.
4. M. Psiuk, T. Bujok, and K. Zielinski. Enterprise Service Bus Monitoring Framework for SOA Systems. *Services Computing, IEEE Transactions on*, 5(3):450–466, 2012.
5. T. Rademakers and J. Dirksen. *Open-Source ESBs in Action*. Manning Publications Co., Greenwich, CT, USA, 2008.
6. J. Zhu, C. Guo, Q. Yin, J. Bo, and Q. Wu. A Runtime-Monitoring-Based Dependable Software Construction Method. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 1093–1100, Nov 2008.