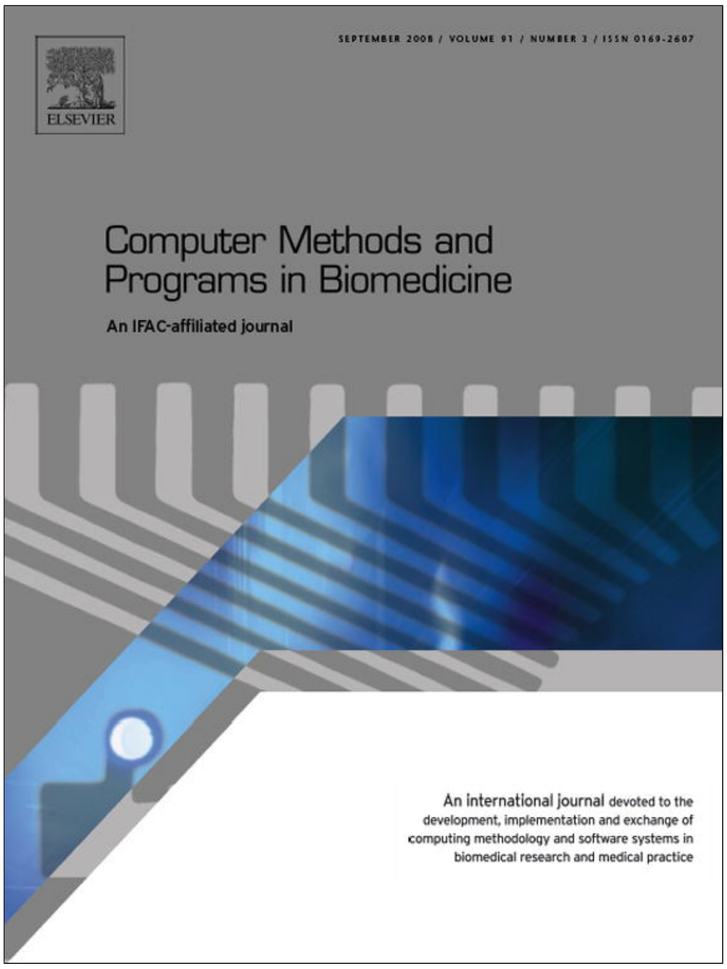
SEPTEMBER 2008 / VOLUME 91 / NUMBER 3 / ISSN 0169-2607
ELSEVIER
Computer Methods and Programs in Biomedicine
An IFAC-affiliated journal
An international journal devoted to the development, implementation and exchange of computing methodology and software systems in biomedical research and medical practice

journal homepage: www.intl.elsevierhealth.com/journals/cmpb

Design of a JAIN SLEE/ESB-based platform for routing medical data in the ICU

Bruno Van Den Bossche[a,*], *Sofie Van Hoecke*[a,*], *Chris Danneels*[b], *Johan Decruyenaere*[b], *Bart Dhoedt*[a], *Filip De Turck*[a]

[a] *Ghent University - IBBT, Department of Information Technology, Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium*
[b] *Ghent University Hospital, Department of Intensive Care, De Pintelaan 185, 9000 Gent, Belgium*

ARTICLE INFO

Article history:
Received 27 December 2007
Received in revised form
10 April 2008
Accepted 16 May 2008

Keywords:
Intensive Care Units
Computer assisted decision making
Software architectures
Computer systems evaluation
High performance computing

ABSTRACT

The importance of computer aided decision making is continuously increasing. In the ICU, medical decision support services gather and process medical data of patients and present results and suggestions to the medical staff. The medical decision support services can monitor for example blood pressure, creatinine levels or the usage of antibiotics. If certain levels are crossed, they raise alerts so that the medical staff can take appropriate actions if required. This significantly reduces the amount of data needing to be processed by the medical staff.

To handle the large amount of data that is generated by the ICU on a daily basis, a platform for routing and processing this data is necessary. In this paper we propose a platform based on JAIN SLEE and an Enterprise Service Bus. The platform takes care of the routing of the data to the appropriate services and allows to easily deploy and manage services. In this paper, we present the design details and the evaluation results. Furthermore, it is shown that the platform is capable of routing and processing all the events generated by the ICU within strict time constraints.

1. Introduction

A computerized Intensive Care Unit (ICU) is an extremely data-intensive environment, resulting in enormous databases. It is generally assumed that every patient generates around 16,000 different data values on a daily base, coming from monitoring devices, laboratory values and manually entered data values by the medical staff. However, processing this large amount of data exceeds human intellectual capabilities [1]. Not only the amount of data, but also the heterogeneity calls for automated data processing in the ICU.

Information technology can facilitate the abstraction of relevant information and can support the physician through software services for medical decision support. A software component qualifies as a service when its business logic is protocol-independent, location-agnostic and contains no state so that the service cannot remember information or keep state from one invocation to another. They do not contain presentation logic, so they may be reused or composed across diverse applications.

Within the ICU of Ghent University Hospital, several implemented medical support services already exist. The RIFLE service will detect kidney dysfunction based on the RIFLE criteria [2]. These RIFLE criteria are an attempt to define Acute Renal Failure (ARF) for critical patients and classify the patient's status according to severity of ARF: Normal, Risk, Injury, Failure, Loss, and End Stage Kidney Disease. The RIFLE service considers the parameters serum creatinine and urine

* *Corresponding authors*. Tel.: +32 933 14900; fax: +32 933 14899.
E-mail addresses: bruno.vandenbossche@intec.ugent.be (B. Van Den Bossche), sofie.vanhoecke@intec.ugent.be (S. Van Hoecke).

0169-2607/$ – see front matter © 2008 Elsevier Ireland Ltd. All rights reserved.
doi:10.1016/j.cmpb.2008.05.003

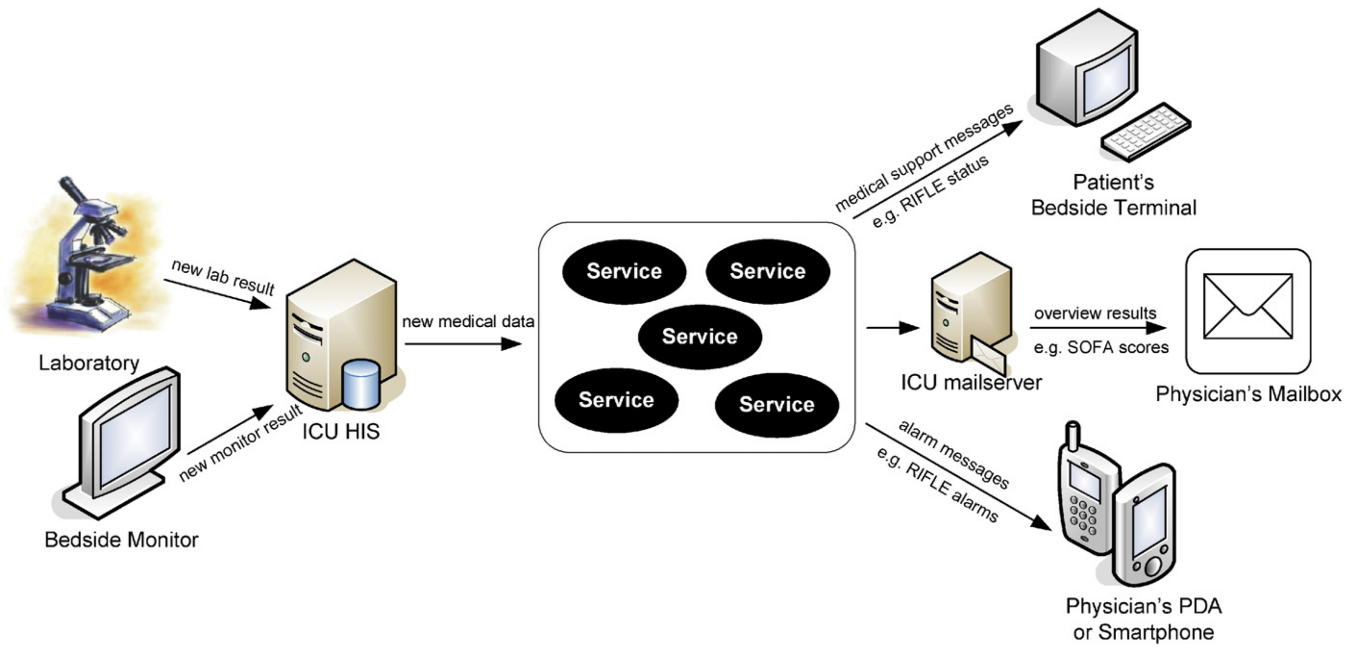

Fig. 1 – Typical flow of data in the ICU with input from monitoring devices, data inserted by staff members or test results which, after processing, result in various output formats such as e-mail, sms or data for bedside monitors.

output where the worst status of these two parameters is taken as the general RIFLE status. Another medical support service implemented in the ICU of the Ghent University Hospital is the SIRS service identifying patients with Systematic Inflammatory Response Syndrome (SIRS) [3] at an early stage, who have risk of progress to severe forms such as sepsis, severe sepsis and septic shock. SIRS is an inflammatory state of the whole body without proven source of infection. SIRS is manifested by the occurrence of two or more SIRS criteria in the patient's laboratory results and monitored data, in a period of 24 h. The SOFA service will calculate daily the Sequential Organ Failure Assessment (SOFA) score [4] for all patients staying in the ICU. This score is used to determine organ dysfunction and organ failure of critically ill patients and mainly used as an outcome prediction for the patient during the stay at the intensive care unit.

It is expected that in future ICU information systems, hundreds of medical support services will be active simultaneously in order to optimize the care of critically ill patients. However, providing a large number of medical support services requires a platform for efficiently routing the medical data to the appropriate services. When new data from monitors or laboratory, captured in the medical HIS databases, is abstracted as events, a stream of events must be routed to the medical support services for processing this medical decision data.

Fig. 1 presents the general setting within the ICU. New data from laboratories or bedside monitors, or data added by the medical staff directly in the medical database, will result in firing an event to the platform, activating interested medical support services for processing these data. The results of the medical support services can be delivered directly to the physician's smartphone or PDA, presented on the patient's bedside terminal or sent by email. For example, a new creatinine value from the laboratory or urine output measurement entered in the ICU Hospital Information System, will result in activating the RIFLE medical support service. If the new values do not impact the patient's RIFLE score, the score is presented on the bedside terminal. If however the RIFLE status changes and potential risk increases, an alarm message is sent to the physician's PDA or smart phone.

Not every physician will or may receive however all the outputs from the different medical support services, only the head of the ICU department can. In order for a nephrologist to receive the outputs of the RIFLE service (a service for detecting kidney dysfunction) the result messages only need to be delivered to the interested physicians, requiring routing of the messages. It would be outside the scope of this paper to review in depth the delivery of service responses and the therefore required message routing, but the authors plan to elaborate on this topic in future publications. In ref. [5] the authors presented an architecture for easy distribution of the medical support services along multiple workstations to execute them simultaneously using Grid technology. This was done by providing run time compilation of medical support service code and service migration. Complementary to that research, this paper focuses on the efficient routing of events from the laboratory or monitors to the medical support services. Fig. 2 shows a schematic example of events which are sent to all the interested medical support services. This way, for example, when a new creatinine value is available, an event is fired and both the RIFLE service and SOFA service receive the new medical data value, while a new billiburine value will result in an event, only received by the SOFA service.

The use of medical support services and computerized systems in the ICU has been gaining momentum and optimally using and visualizing data has been an important research topic in the recent years [6]. Going from information systems

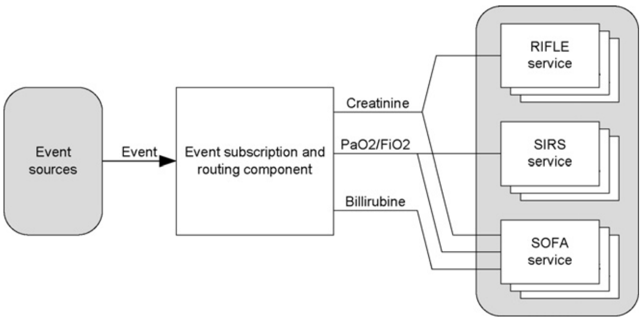

Fig. 2 – Schematic example of the possible event routing of data to medical services.

to assist in the management the medication of patients [7] to the evaluation and development of improved user interfaces for the medical staff [8]. These topics are not the focus of this paper, but it aims to provide a back end software architecture which is suitable for the routing and aggregating of medical data which can be integrated with these solutions.

The remainder of this paper is structured as follows: Section 2 details the functionality a medical decision support platform needs to offer. An overview of the current state of the art and available technologies is given in Section 3. In Sections 4 and 5, a detailed description of the Architecture of the proposed platform and the implementation is given, followed by a thorough performance evaluation of the platform in Section 6. A general discussion is provided in Section 7 and gives an overview of the current research in the field, related work and possible alternative approaches for building medical decision support service platforms. Finally, we highlight the main conclusions of this work in Section 8.

2. Functional requirements

It is generally accepted that within the near future, Intensive Care Unit computerization including advanced real time and bed-side decision support capabilities, will become essential to guarantee the quality of care for every ICU patient. The ICU of Ghent University is the second largest ICU in Belgium and currently holds 56 computerized beds. It is expected that in future ICU information systems, tens or even hundreds of medical support services will be active simultaneously in order to optimize the care of critical ill patients [9]. Scalability of the platform is thus required to handle the message routing for hundreds of simultaneous medical decision support services. The platform should also be generic in order to be independent of the implementation language of these medical support services.

The RIFLE service is a typical medical support service which will be running for each patient and will calculate potential risk for kidney failure based on serum creatinine and urine output as variables. Whenever new creatinine or urine data is available, RIFLE results should be instantly delivered to the physicians. Transparent data routing of new laboratory results or monitor data in the HIS database is thus a requirement.

Currently the RIFLE service is running for the 56 computerized patients and generates at most 678 triggers a day, resulting in an average of 12 triggers per patient, ranging from patients with 4 triggers per day to patients with 27 triggers per day. Other medical support services can however have higher trigger frequency. The SOFA service is for example only triggered once a day for a new billirubine value, but every 5 min a new blood pressure value triggers the SOFA service. Considering all the SOFA parameters, the SOFA service generates on average 465 triggers per patient per day. The SIRS service even exceeds this and generates around 3000 triggers per patient per day. Since every patient generates on average 16,000 different data values on a daily base, each of these data values can become a potential trigger in the future when more medical support services are deployed in the platform, resulting in at most 16,000 triggers per patient per day. Next year, the ICU of Ghent University will migrate two more units to the platform, bringing the total to 94 beds in the computerized ICU or a maximum average of 17 triggers per second for an ICU of 94 patients and 16,000 triggers per patient per day. These triggers can be seen as an event that might be useful to one or more medical support services for processing this medical decision data. The tons of events obviously require a platform that is scalable and has good performance. The platform also has to meet the stringent requirements imposed by an ICU. The routing overhead imposed by the platform must be small enough to be negligible compared to the execution time of the medical support services since delays in trigger delivery will result in delays in treatment actions, which can, especially in critical care medicine, have important negative impacts on outcome.

Within the platform, triggers need to be able to activate one or more medical support services. The results from these services are not sent back to the initiators, but forwarded to the subscribers. As a consequence traditional request brokers cannot fulfill the requirements and a new platform is designed and presented in this manuscript.

3. State of the art

Currently only 10% to 15% of Intensive Care Units are computerized [10]. The Intensive Care Unit of the Ghent University has started computerizing their department in 2003 in order to result in a paperless ICU by capturing and storing all data from monitors, ventilators and pumps in databases. Communication among devices and the HIS database is done by proprietary communication over RS-232, but a standard such as ISO/IEEE 11073 [11] or HL7 [12] can also be used. Up till now, the status of available clinical decision-support systems continues to change and implementations are making strides [13–15]. However they are not providing any means to efficiently integrate developed algorithms, scores and tools for medical decision support [15–18]. Also the addition of new medical decision rules and alerts within these systems is slow and difficult and they use the same medical decision rules for all patients [15], resulting in limitations for physicians to interact with the electronic alerting systems. To the authors' knowledge, a medical support platform able to handle the large amount of data generated by the ICU and overcoming the shortcomings of current solutions, has not been reported upon yet and is the subject of this manuscript.

A number of currently available technologies have been considered to serve as a starting point for implementing a medical support platform. One of these technologies is JavaEE, a component based technology targeted to business applications which offers a container for applications to be deployed and run in. The main function of the container is the life cycle management of the deployed applications, i.e. starting and stopping the application or application components. Furthermore, a number of so called non-functionals are delegated to the container instead of the actual application. These non-functionals include logging, authentication, management of external resources, etc.

The application container hosts JavaEE applications composed of Enterprise Java Beans (EJBs). There are three type of EJBs: Session Beans which contain the business logic, Entity Beans which represent data and Message-Driven Beans which allow for asynchronous communication. Additionally a web-container is provided which enables the hosting of HTTP-based services.

The typical way of communicating with a JavaEE Application server is through synchronous method calls and the same goes for the inter component communication. Clients can interact directly with application components using a JavaEE application client which performs Remote Method Invocations. For asynchronous communication the Java Message Service (JMS) [19] can be used in combination with Message-Driven Beans (MDB). JMS allows to configure communication channels called queues (one-to-one communication) or topics (one-to-many communication), MDBs consume and process the incoming messages on those channels. However, the capabilities of the JMS system are somewhat limited as the queues and topics are statically configured at deploy-time and no other efficient routing mechanisms are available in the framework. This makes JMS suitable for implementing asynchronous business flows as shown in ref. [20], but less suitable for low latency event processing. Performance evaluation results of JMS queues [21] show that the average latency introduced by JMS varies between 29 and 139 ms for transactional messaging and message sizes of 1 KB excluding network latencies. We do not consider these timings to be strict enough as a trigger might result in several events which would significantly increase the response time of the medical services.

Another competing technology is Web services. In view of the broad support for Web services and common XML-based standards, Web services are a promising concept for the integration of heterogeneous software components. By means of this technology, applications can easily be distributed and expose well-defined functionality as a Web service, which consumes and produces XML-messages over HTTP. Based on the exchange of structured text messages, the interaction makes abstraction of the underlying technologies.

Web services support both synchronous and asynchronous communication between the client application and the actual service. Asynchronous message passing may improve system usage and avoid delays on the client side, while waiting for the Web service results, but is however more difficult to code and can introduces several problems. As an example, the calling process does not wait for delivery of the message, and thus never hears about possible errors. In order to discover the completion of the called function, the application will either have to create a polling mechanism, event trigger, or callback method in order to be later notified of the operation. Consequently, the event-oriented platform using Web service technology must not only do event routing, but also implement event subscription. Different event types can be specified as message part from a single Web service operation, or can be mapped each on different Web service operations. In both approaches however, any changes to the events or new events added to platform require modification of the Web services.

The Web services architecture now supports this through a lightweight eventing protocol. WS-Eventing is an existing specification that became a W3C Submission in March 2006 and that allows Web services to subscribe to or accept subscriptions for event notification messages [22]. This specification defines a protocol for one Web service to register interest with another Web service in receiving messages about events, events being SOAP messages. The subscriber may manage the subscription by interacting with a Web service designated by the event source. There are many mechanisms by which event sources may deliver events to event sinks. This specification provides an extensible way for subscribers to identify the delivery mechanism they prefer.

4. Platform description

As already stated, medical decision data is constantly being generated in the ICU. The generation of every new data value generates an event that might be useful to one or more medical support services for processing this medical decision data. In this section we propose a software platform which enables and simplifies the automated decision making in the ICU. First we give an overview of the capabilities such a platform is expected to offer. Next the actual platform is discussed in detail.

The Intensive Care Unit of the Ghent University captures and stores all data from monitors, ventilators and pumps in HIS databases by using proprietary RS-232 communication among devices and the HIS database. Capturing new data in the HIS needs to result in generating one or more events. Processing the medical decision data is twofold, first the data needs to be received and routed toward its destination, i.e. the medical decision making service, next the actual processing needs to be done by the service.

Fig. 3 presents the proposed platform to be used in the context of routing and processing of medical decision data. It consists of two main parts: a bus-like communication platform (labeled Message Platform) and an application platform which can host and execute the medical services (labeled Service Container).

One of the key features of the platform is that it supports event-based services. As already detailed in Section 2 there is no direct request-response type pattern in processing the data. When a device or monitor inputs data in the HIS database, an event is generated in the platform. After a data message has been inserted in the system it will be evaluated, processed by a medical service, and maybe grouped or related to other data, resulting in a new event to be pushed to an output device such as the PDA of a physician. The device or monitor however does not expect a response. The output

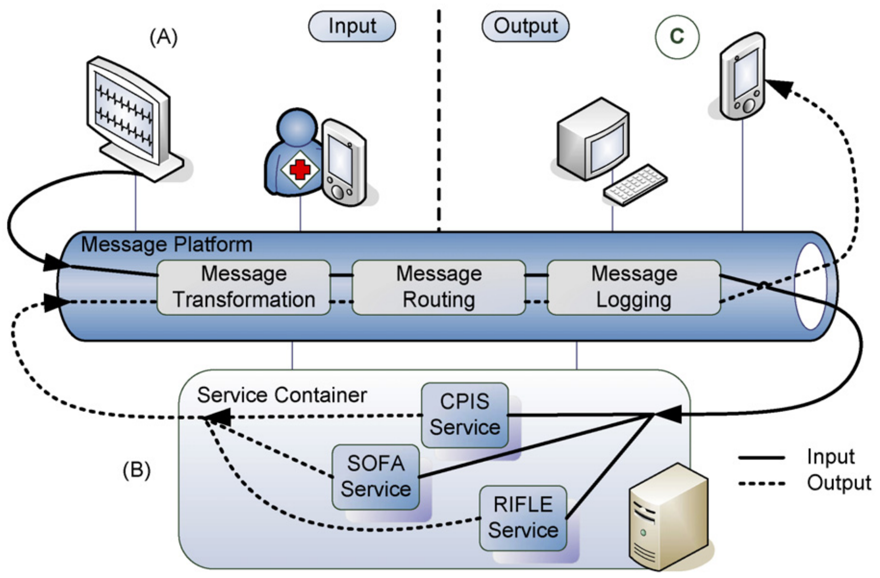

Fig. 3 – Global architecture of the platform consisting of a Message Platform which is responsible for routing and transforming input and output data and a Service Container which hosts all services and event processing services and is responsible for the inter-service routing of messages.

of one service might also be used as input for an other service which allows to chain and combine multiple services into one meta-service. Similarly can a data message inserted into the system be used as input for a number of (independent or parallel) services.

The typical message flow in the system is as follows: A monitoring device (or an interactive interface for the ICU staff for that matter) generates new data in the HIS database, resulting in an event (see Fig. 3 part A) pushed into the message platform. The transformation component of the message platform translates the data into a usable format, the routing component redirects it to the application server. Additional intermediate components such as a logging service can be invoked as well. In the application server the message is provided automatically to all service components which require this type of data as input (see Fig. 3, part B). The services processes the data and generate new events or output data. This data can in turn be fed to other services or could be sent to output devices. These include bed side monitors, terminals in the ICU or mobile devices of physicians or ICU staff (see Fig. 3, part C). Routing of the output messages is handled by the routing component in the message platform.

It might not be desirable to forward messages toward any output devices, for example not every staff member should be able to see all patient data. Going into detail of these additional requirements is out of scope for this work, but solutions such as rule based filtering can easily be implemented on top of this platform by inserting an extra routing component inside the message platform. The platform supports the deployment of additional components which can be used to check for each message if the required authorization for each recipient or sender are available.

In the current platform the ICU gathers data from the devices and stores the information in a database. A prototype, where updates in the database are used as a trigger to generate the events in the platform, has been implemented and is currently being evaluated by the Department of Intensive Care of the Ghent University Hospital. If however the monitoring devices are capable of generating events and transferring them over the network, these events can be pushed directly into the platform. Another approach would be to make use of auxiliary devices deployed next to the monitoring devices which can take on the task of transforming monitoring data into suitable events that can be transferred over the network. It might even be relevant to insert data from other non-medical devices into the system (for example the room temperature, the amount of light in a room, etc.). As a consequence, any platform to be used in the context of an ICU should be extensible to simplify adding new devices and possibly translate incoming data into a format already known by the system.

5. Implementation details

Based on the stated requirements for the proposed platform we evaluated currently available technologies capable of handling a large amount of events. Next the technologies should as well be capable of interfacing with a large number of different devices and specific protocols. We will first discuss two enabling technologies, followed by a more detailed explanation of how they are used in the implementation.

5.1. *ESB overview*

An Enterprise Service Bus (ESB) is an architectural pattern which defines a communication infrastructure between services. Providing an exact description of an ESB proves quite difficult as definitions vary from source to source. However, most definitions do agree on the most important characteristics.

An ESB is based on standards and acts as a messaging system or messaging backbone between different services. Instead of letting services communicate directly with each other, they interact with the ESB which automatically routes all messages to their destination and if necessary translates them to a format the destination endpoint understands. It is essential that the ESB supports a wide variety of protocols and offers the facilities to easily translate messages from one protocol to another. This allows to remove any coupling between calling a service, the message protocol used and the required message format. The main role of the ESB is to serve as a communication bus accepting a variety of input message formats and capable of transforming these messages to different output formats and thus providing a transparant communication interface.

This means that an ESB implementation itself is not standardized but offers a messaging infrastructure based on standardized protocols. As a result, there are major differences in the feature sets of available ESBs as vendors try to differentiate from each other. A functional overview of commercial ESBs is presented in [23,24] evaluates the performance of four both commercial and non-commercial ESB implementations.

5.2. *JAIN SLEE overview*

JAIN SLEE is the specification of a component-based container for high-throughput asynchronous event processing and is part of the JAIN initiative (Java APIs for Integrated Networks) [25]. The JAIN initiative defines a set of Java technology APIs that enable the rapid development of Java based communications products and services. The JAIN APIs are currently mostly used to implement telecom related applications, but there are no reasons to *not* employ it in different contexts. SLEE stands for Service Logic Execution Environment and is a well-known concept in the telecommunications industry. It encompasses a framework that allows to rapidly develop and create complex services. Next, it allows a straightforward composition of basic services into more complex services without an additional development effort.

The JAIN SLEE container is comparable to the JavaEE container [26] and thus has much the same advantages such as the life cycle management and provisioning of the non-functionals. There are some important differences however. Instead of EJBs the components are called Service Building Blocks (SBB) and the internal communication is completely event-oriented and asynchronous. And it is exactly this event-oriented nature of JAIN SLEE that is useful to handle the large amount of events generated in the ICU. Instead of implementing certain interfaces, SBBs define the types of events they can consume and produce in a deployment descriptor. When an event is generated it will automatically be routed to all components which are capable of consuming this type of event. After consuming an event, an SBB may generate a new event which again will be routed automatically to the right consuming components. The main role of the JAIN SLEE platform is to automatically route events between different deployed medical services. The advantage of using JAIN SLEE is that this logic is performed entirely by the application container and requires no additional work from the service developer. Deploying new medical services will integrate them automatically in the event flow. This significantly reduces the management overhead for the platform.

There is no standard protocol to communicate with the JAIN SLEE container but any protocol can be supported by plugging a Resource Adapter (RA) into the container. This RA then translates incoming messages to Events understandable by the container and vice versa. Furthermore, it will assign all related events to a Session, called an Activity Context. This means that events which are part of a logical event-flow (e.g. coming from a single patient) can make use of each others context-information. This approach was chosen by the developers of the JAIN SLEE specification to allow JAIN SLEE to be an extensible and protocol agnostic application container.

5.3. *Platform implementation details*

As explained in Section 4, the main platform components are: message transformation, message routing and the actual service components. Additional components such as logging can be integrated in the platform as well. The same applies to JAIN SLEE Resource Adapters to support additional protocols or communication with custom external systems. Of these components, message routing and logging are services offered by both the ESB and JAIN SLEE and can be used as is. In case of the ESB the routing logic is managed by an XML-descriptor which statically connects different transformers to input and output channels. For JAIN SLEE, the routing is performed fully automatic based on the type of input events and output events in the XML-descriptors of the services.

Plate 1 gives an example description file of a typical SBB implementation which has one type of input events and one type of output events.

The message transformation, service components and RAs all have a different implementation complexity. Message Transformation converts messages from one message type to another. The implementation requires one *transform()* method containing the translation logic. The transformation includes the conversion from one data format to another and tagging each message with a patient ID which allows the JAIN SLEE to logically work with input from patients instead of from devices. An in memory data store is maintained which links devices to patients and vice versa. The overal complexity of the transformers is low.

The implementation of the JAIN SLEE service components requires an *onEvent()* method for each event-type the service wishes to receive. The implementation complexity is largely determined by the complexity of the medical service. RAs are the most complex components of the architecure. The RA handles all incoming and outgoing messages of the JAIN SLEE platform and needs to be designed with great care to prevent becoming the bottleneck of the application. Low level interaction with the JAIN SLEE application server is required and thus a very good knowledge of the platform is a must. The Resource Adapter reads incoming messages and converts them into events suitable for processing in the JAIN SLEE Application Server and maintains the patient-oriented sessions, called Activity Contexts. The implemented service consists an SBB which uses the Activity Context to store session information and updates this information on every event. Using these ses-

```
<?xml version="1.0"?>
<!DOCTYPE sbb-jar PUBLIC
 "-//Sun Microsystems, Inc.//DTD JAIN SLEE SBB 1.0//EN"
 "http://java.sun.com/dtd/slee-sbb-jar_1_0.dtd">
<sbb-jar>
  <sbb id="MService1">
    <description>
      Medical Service 1 Implementation
    </description>
    <sbb-name>MSI</sbb-name>
    <sbb-vendor>IBCN</sbb-vendor>
    <sbb-version>1.0</sbb-version>

    <sbb-classes>
      <sbb-abstract-class>
        <sbb-abstract-class-name>
          be.ugent.ibcn.MedicalService1
        </sbb-abstract-class-name>
      </sbb-abstract-class>
    </sbb-classes>

    <event event-direction="Receive" initial-event="True">
      <event-name>InputEvent</event-name>
      <event-type-ref>
        <event-type-name>InputEvent</event-type-name>
        <event-type-vendor>IBCN</event-type-vendor>
        <event-type-version>1.0</event-type-version>
      </event-type-ref>
      <initial-event-select variable="ActivityContext" />
    </event>

    <event event-direction="Fire" initial-event="False">
      <event-name>OutputEvent</event-name>
      <event-type-ref>
        <event-type-name>OutputEvent</event-type-name>
        <event-type-vendor>IBCN</event-type-vendor>
        <event-type-version>1.0</event-type-version>
      </event-type-ref>
    </event>
  </sbb>
</sbb-jar>
```

Plate 1 – XML-configuration example of a JAIN SLEE SBB which implements a service and defines the type of events this component takes for input and produces for output.

sions allows to store state information on a patient base. A typical medical service does not maintain state (i.e. subsequent invocations do not share data). However, it is relevant to maintain state information on a per patient basis. For example in case of the RIFLE service different parameters are monitored, and it is beneficial to maintain the last measured value of each parameter. On each update of such a parameter, the service can be executed and the necessary parameters values are available through an in memory per patient session object. This allows a much more efficient execution compared to a service that needs first needs to fetch all relevant data from an external data store.

Although the RA is the most complex component, there are only a limited amount of resource adapters required, usually even only one, namely the RA responsible for the communication between the ESB and the JAIN SLEE container. Message transformation to a unified event type can be taken care of by the ESB. Additional RAs might be required to interact with other external systems unrelated to the medical monitoring data, for example a database for information storing or event logging. However, these type of RAs are typically already made available by the JAIN SLEE vendors. For all components IDE-tools to aid with the implementation and creation of the necessary descriptors and configuration files are available.

For communication between the ESB and the JAIN SLEE platform UDP was chosen in favor of TCP because of the lower overhead and its simplicity. As UDP is a connection-less and unreliable protocol the loss of messages has to be taken into account and a simple retransmission protocol is implemented. However, as the framework is to be deployed in a closed network with limitid or no packet loss the overhead of the retransmissions will be minimal. TCP could be used in the implementation as well, but would result in a decrease in performance.

5.4. *Scalability and distributed execution*

The scalability as such may not be the main problem in the context of an ICU as the number of events will be significantly lower than the event count in telecom operator grade services where technologies as JAIN SLEE were initially developed for. However, the robustness of the platform should be of the same high quality. Hence a distributed execution of the platform can help in obtaining this goal. First of all, both the available ESBs as JAIN SLEE implementations are developed with a distributed execution in mind. The platforms can be deployed over multiple server-nodes with fail-over capabilities, i.e. if one node fails, other nodes will handle the requests to these nodes and recover the ongoing transactions using a session replication mechanism. Due to the combination of the ESB and JAIN SLEE platform it is also possible to deploy these platforms on different systems where the ESB could function as a load balancer between JAIN SLEE nodes. Using the built in capabilities of both platforms and using the ESB as a load balancer, it is possible to offer a robust platform with fail-over mechanisms.

6. Platform evaluation details

To evaluate the platform we investigated the behavior and performance of both the subcomponents of the architecture and the platform as a whole. First we present an overview of the test setup and the obtained results of the performance evaluation of the individual components, followed by a discussion and comparison of the results of the integrated platform.

6.1. *Test setup*

All tests were performed on a AMD athlon XP1600+ with 256MB of memory and an installation of Debian GNU/Linux with kernel 2.6. The server is connected to the load-generator using a 100 Mbit switched network. This modest hardware platform was chosen explicitly to demonstrate the low system requirements of the platform. This proves it is possible to host the hardware very close to the actual ICU instead of a specialized hosting facility in the hospital which simplifies the implementation and integration in the ICU. For example this makes it possible to deploy a server running the platform in a nursing station where it benefits of all facilities of the ICU department such as guaranteed power supplies.

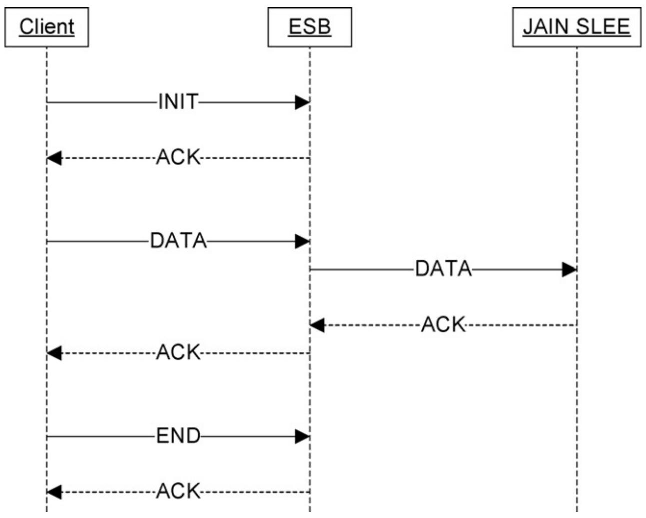

Fig. 4 – The typical input scenario used in the performance evaluation.

To simulate clients and collect time measurements the load testing tool Apache JMeter [27] was used. JMeter is a free Open Source extensible load generator which allows to simulate a large amount of clients and gather time measurements. It features dynamic display of statistics about running tests such as the event rate and the round rip delay and allows to save these results in a csv or xml file for offline processing. The system under test was running a single-node JAIN SLEE provided by OpenCloud (RhinoSDK) [28] and an instance of the Mule ESB [29] executed on top of the Sun JDK 1.5. A typical scenario as shown in in Fig. 4 replicates the behavior of a device inputting data in the system. First an INIT message is sent, next the actual data is inserted and processed by the system. After processing the acknowledgment with output data is sent back and finally the input-session is terminated with an END message. Each message is acknowledged by the service, and the response time is the time measured from the INIT messaged until the acknowledgment to the END message. This means that for every data value inserted in the system, three events are sent. For measurement purposes the initial event and output result of a service invocation are routed through the same workstation as this allows us to perform accurate time measurements on the same clock. In real world deployments the initiating device and output device are likely to be different. The INIT messages are generated using a uniform distribution and all following messages are sent immediately after the acknowledgment of the previous message.

6.2. *Platform performance tuning*

The technologies in this paper are all Java based and an important feature of Java is the use of a garbage collector. Java includes automatic memory management (garbage collection) as a part of the Java runtime. This means that very common errors made by developers related to memory management cannot occur. One of the side effects of the garbage collection is that the application execution can be paused, at unpredictable times, to allow for garbage collection. These pauses

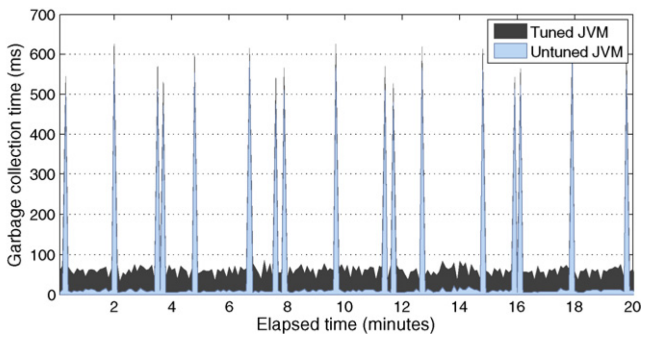

Fig. 5 – Without tuning the garbage collection pauses the JVM completely when garbage collection occurs with pauses regularly of over 500 ms. With tuning the average pause is higher, but the garbage collection behavior is much more stable and predictable and partially occurs without pausing the JVM.

are highly undesirable when dealing with low latency and critical services. However, by setting specific parameters of the Java Virtual Machine it is possible to prevent long pauses introduced by the garbage collector, although at the cost of a small performance penalty. With appropriate tuning, these pauses can be minimized and thus the obtained results significantly improved. The result of tuning the JVM is shown in Fig. 5 which is a 20 min snapshot of a test run once with tuning and once without tuning. Without the tuning, the time spent in garbage collection is lower in the majority of the garbage collector invocations, but at regular intervals a very long garbage collection takes places which pauses the application execution. With tuning the time spent in garbage collection is on average longer, but the result is much more predictable and no long pauses occur. It should also be noted that the tuned garbage collection times are the accumulated times of the parallel and concurrent collectors which are partially executed concurrently with the application without pausing the application execution. In contrary to the untuned garbage collector which always pauses the application execution.

The set of tuning options used for all platforms is shown in Table 1. Detailed results of JVM tuning were previously reported on in ref. [30]. The memory managed by the virtual machine is divided into multiple generations (Young, Tenured and Perm), depending on the age of the objects. As objects live longer they are moved into the next generation after a certain amount of time or a number of garbage collections. By specifying the sizes of the generations (1–3) and limiting the amount of time before an object is promoted to the next generation (4–5), we can achieve that objects lasting for the duration of a whole session to move to the Tenured generation very fast. This is beneficial as the older generations do not need to be garbage collected as often since the the majority of objects die very young. The garbage collector itself can also be tuned (6–11) to use multiple threads and to work concurrently with the application execution for as long as possible. Using a concurrent garbage collector significantly reduces the length of the pauses that stop the application execution. Tuning of the JVM allows to limit the time the execution of the virtual machine needs to be paused completely for garbage collection and makes the garbage collection more predictable. An in depth discussion of the JVM tuning options can be found in ref. [31].

Table 1 – Virtual Machine Tuning options for low latency behavior

-Xm×128m	(1)
-XX:NewSize=32m	(2)
-XX:MaxNewSize=32m	(3)
-XX:MaxTenuringThreshold=0	(4)
-XX:SurvivorRatio=128	(5)
-XX:+UseParNewGC	(6)
-XX:+UseConcMarkSweepGC	(7)
-XX:+CMSIncrementalMode	(8)
-XX:+CMSIncrementalPacing	(9)
-XX:CMSIncrementalDutyCycleMin=0	(10)
-XX:CMSIncrementalDutyCycle=10	(11)

6.3. *Performance results*

This section presents the obtained results of all performed tests. The Mule ESB and Rhino JAIN SLEE were tested separately first and these results are shown in Fig. 6. The test scenario in this case is a simplified case of the scenario shown in Fig. 4 where there is only one system under test and 200 emulated devices submit updates to the system. For Mule the minimal functionality of a medical service was implemented to replicate the same behavior of the JAIN SLEE solution. This additional test provides us with info about the maximal throughput of each individual system. On the x-axis the sustained event rate is plotted and on the y-axis the response time. Additionally the average system load during the test run is plotted as well. Each measured event rate was sustained for 1 min and preceded and terminated by a ramp-up period of 1 min, all measurements were executed sequentially this way without restarting any of the applications.

We notice that the measured response times remain very low during the test, even at the highest event rates for Mule (Fig. 6(a)) contrary to a standalone JAIN SLEE application container (Fig. 6(b)). The JAIN SLEE test clearly shows that the platform performs significantly slower than the Mule ESB regarding latency and the maximum throughput that can be obtained. One of the main bottlenecks is the number of simultaneous threads or simulated client devices. Each device requires a new session to be set up whenever it inserts data into the system. Additional results shown in Table 2 show that with less simultaneous devices JAIN SLEE performs significantly better. So, contrary to Mule, the behavior of the JAIN SLEE application container is much more affected by the number of input-devices. This is to be expected as the session model within JAIN SLEE is much more complicated and versatile compared to the session model in Mule.

Combining both platforms results in lower latencies and higher achievable event rates than is the case with a JAIN SLEE only solution as shown in Table 2 and Fig. 7. As expected the latencies are slightly higher than with Mule as the messages have to pass through two separate application containers, however they are lower than with a standalone JAIN SLEE.

Table 2 – Performance results for a varying number of monitoring devices showing the obtained measurement results at 50% system load and the maximum obtained event rate

Platform	Devices (#)	Event rate (events/s)	Average (ms)	95th (ms)	99th (ms)	Maximum event rate (events/s)
Mule	10	325	5	10	19	550
JAIN SLEE	10	150	26	127	215	375
Combined	10	155	14	33	128	325
Mule	100	300	6	12	23	475
JAIN SLEE	100	145	31	197	314	250
Combined	100	155	15	37	158	295
Mule	200	275	9	15	30	450
JAIN SLEE	200	140	33	206	331	240
Combined	200	150	17	39	206	280

Increasing the number of simultaneous devices most significantly impacts JAIN SLEE as a stand alone platform, but does not have such a large impact on the combined platform which offers the best functionality of all.

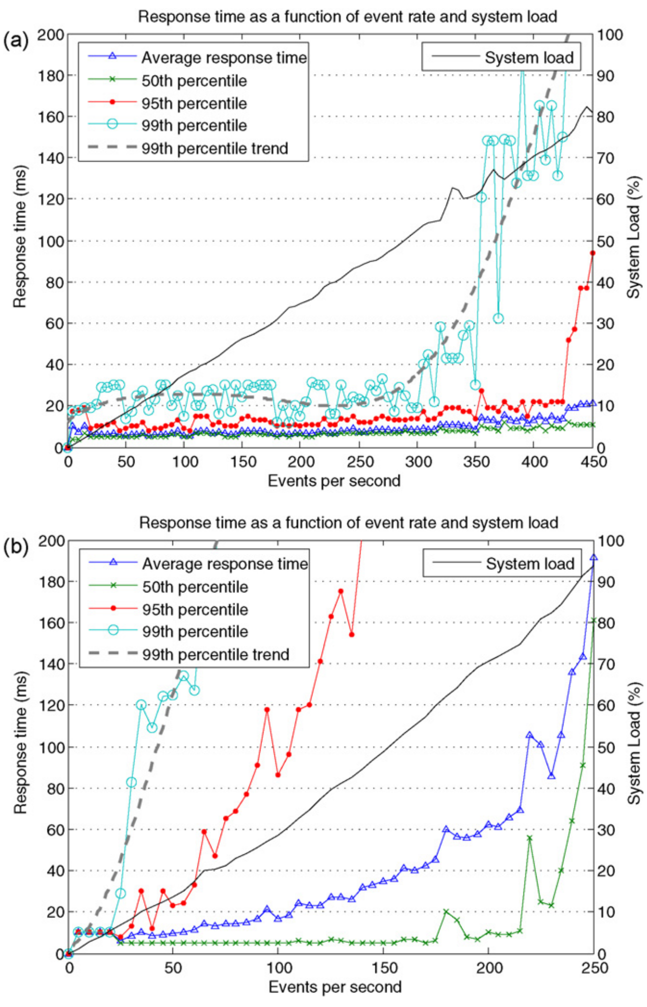

Fig. 6 – The obtained performance results for the different scenarios. (a) Response times for the Mule ESB when 200 devices continuously submit data entries into the system at different event rates. The response times remain low until the maximum event rate is achieved. (b) Response times for the Rhino JAIN SLEE implementation when 200 devices continuously submit data entries into the system at different event rates. The response times significantly increases as the event rate increases.

Important to note is that the number of client devices has a very limited influence on the measured latencies.

Overall the obtained results for the combined platform are better than with JAIN SLEE as a stand alone solution. It is still less performant than a pure Mule solution, but it does offer far superior functionality for developing and implementing complex services.

6.4. *Platform overhead evaluation*

To be able to illustrate the performance results of the developed platform, an equivalent implementation of the test service was recreated as a standalone application. Comparing the results of the standalone version of an equivalent service to the results of the service deployed in the framework allows us to determine the overhead imposed by the framework.

As to be expected is the performance of this implementation better than when using a framework for this

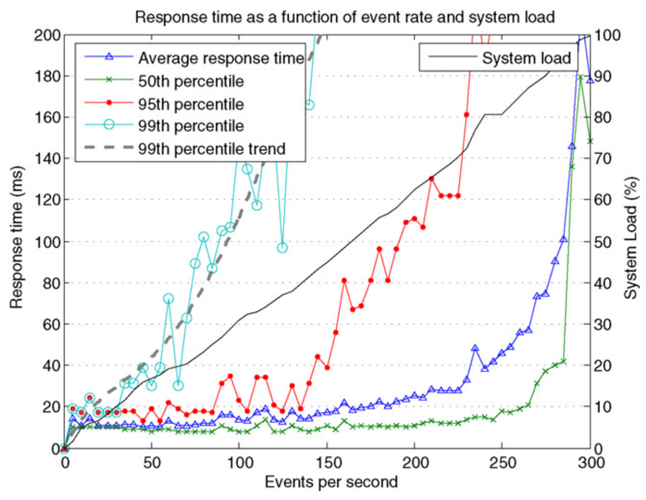

Fig. 7 – Response times for the combination of the Mule ESB and Rhino JAIN SLEE implementation when 200 devices continuously submit data entries into the system at different event rates. The performance and especially the response times are significantly better than a standalone JAIN SLEE implementation.

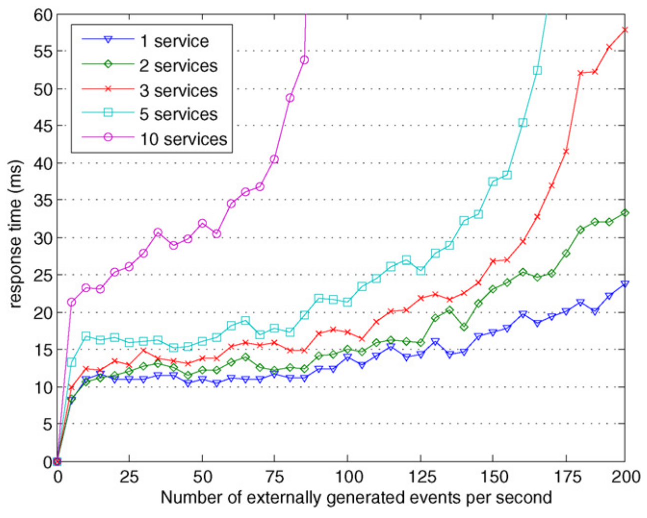

Fig. 8 – Measured latency of multiple chained services. The latency latency increases less than linearly for each additional service as the overhead for accepting and processing an external event is larger than for processing internal events which are sent between the different services.

straightforward test case. In the test with 200 simulated clients, the maximum obtained event rate reached 1000 events per second and the measured latency was on average 5 ms which is more than 3 times with the combined platform solution. This is however the special case of only one single service. Additional tests include the case where multiple services are deployed in parallel, i.e. multiple services use the same input-events, or are chained after one another, i.e. services use results from other services. The same tests were performed for 1, 2, 3, 5 and 10 services chained after one another and the obtained average response times are shown in Fig. 8.

As expected the latency increases as the number of chained services increases. However, the increment size is less than the latency of one single service. This can be attributed to the limited overhead that an additional service causes, the only extra delay is caused by the time it takes to execute the service, but the application container does not have to spend as much time to set up sessions, parse the incoming messages, etc. This initial overhead occurs for every message entering the application container and activating a service to be executed. Executing additional services once inside the application container and in an existing session only adds a much smaller overhead.

In the case of 10 services being chained, a event rate of approximately 75 events per second can be obtained and for 5 chained services this is approximately 150 events per second. Although the achievable event rate halves as the number of services doubles, the actual amount of events processed in the application container remains the same. In the case of standalone services the execution time increases linearly with the number of services. In the case of 10 chained services the total execution time is 50 ms compared to the 30 ms if the services are deployed in the platform. For one type of triggers, needed to be processed by one service, the platform imposes an additional overhead. However, if output of services needs to be chained together (i.e. the output of one service functions as a trigger for another service) or triggers are processed in parallel by multiple services, the platform outperforms standalone implementations of these services due to the efficient routing of events inside the platform.

7. Discussion

ICU medicine is expensive and there is a shortage of intensivists. The introduction of computerized decision support is one of the most important adaptations to improve quality of patient care for the near future. It is important that physicians only receive alerts of patients they are responsible for and not for all ICU patients so that every alert is meaningful. For the RIFLE medical support service, receiving the RIFLE alarms was not only accepted by the ICU-physicians of the Ghent University hospital but even appreciated. Further research is currently performed to investigate how much faster therapeutic intervention is induced by real-time RIFLE alerts, and if this faster intervention leads to better preservation of kidney function and better patients' outcome. As stated before, it is expected that hundreds of medical support services will be active simultaneously in order to optimize the care of critically ill patients. Delays in treatment actions, especially in critical care medicine, can have important negative impacts on outcome. By using our platform it is possible to move from discovery of information to anticipation through delivering support and alarm messages.

In our opinion, combining JAIN SLEE with an ESB is the best choice for implementing decision support systems in the ICU since the obtained results for the combined platform are better than with JAIN SLEE as a stand alone solution. It is however less performant than the standalone ESB solution, but it does offer far superior functionality for developing and implementing complex services. However, the platform can be more performant than custom developed stand alone services. As soon as services are being triggered on the same input events or are chained together, the response time is actually lower for services deployed into the combined platform. Another advantage of using currently available technologies is that we can take advantage of already built in capabilities regarding security. Additionally current research focusing on policy enforcement can for service enabling platforms such as JAIN SLEE and ESBs [32] could be added to the framework transparently. This enables the platform for example to add rules that only allow a patients doctor to receive certain medical data, or to automatically send alerts to his smartphone.

Alternative technologies such as J2EE and Web Services were considered but these were currently found less suitable to serve as base components for the medical support platform. Within the event-oriented ICU platform, the routing of events is crucial to the success of the platform and the lack of an efficient and easy to use event model within JavaEE hampers its usability. A third party or custom built event mechanism can be plugged in to the JavaEE Application Server using JavaEE Connector Architecture Resource Adapters (RA)

[33,34]. This would enable JavaEE to replace JAIN SLEE in the current architecture, but this also incurs a very high additional and customized development cost with higher maintenance costs as not only and additional RA is required but also the use of non-standard service components. Because of this reason JAIN SLEE was given preference as it offers superior routing capabilities and the required functionality is available out of the box. Yet the application server can still be extended using RAs if required.

The use of Web services could be a viable option. Within the event-oriented ICU platform, the event types can be mapped to event sources where the Web services can subscribe to receive these events. However, at the time of writing the available support for WS-Eventing is still lacking. The necessary tooling and support from vendors is currently lagging behind on the support for other technologies. In the current architecture it would be fairly simple to integrate Web service based services as the ESB can translate SOAP-messages into other formats, such as events suitable for processing by JAIN SLEE or it would even be possible to replace the JAIN SLEE application server with a Web services environment, maintaining the ESB to provide communication capabilities to non-Web service enabled devices.

8. Conclusions

In this paper we presented a platform to enable the development and deployment of computer aided medical decision services. The goal of this platform is to offer a generic platform on top of which complex services can be deployed to be used for computer aided decision making in the ICU. Key features are the support for event based applications, the extensibility of the platform to support any type of medical or non-medical input-device. In order to achieve this, a hybrid architecture is designed which consists of an Enterprise Service Bus and a JAIN SLEE container. Benefits of the ESB are that it is a very suitable technology for routing messages and converting them into other appropriate formats. The JAIN SLEE Container is specifically developed to simplify the development and deployment of event based services. Using the platform it is very simple to translate the output of medical services to messages suitable for a large variety of output devices such as bed-side monitors, PDAs, DECT or mobile phones.

To validate the platform, a thorough performance evaluation was performed. The obtained results clearly show that by using the hybrid architecture it is possible to sustain and process a large stream of events and still guarantee low response times. It is also shown that the platform outperforms a combination of custom built stand alone services. Using the hybrid architecture also simplifies the development and deployment of new medical services as the majority of required non-functional services (i.e. transaction management, logging, life-cycle management) are already provided by the application containers.

Summary points

What was already known:

- More and more medical support services will be used to monitor and optimize the care of critically ill patients.
- Computer aided decision making is increasingly important in the ICU and will soon become a necessity due to the vast amounts of data to process.
- The lack of a generic versatile platform hampers the deployment and integration of multiple and diverse medical services in the ICU.

What this study added:

- An evaluation of currently available technologies to determine the suitability for implementing a medical decision support services enabling platform.
- An extensible software platform based on currently available technologies to simplify and optimize the deployment and usage of event based medical decision support services.
- A thorough performance evaluation of the platform to validate the performance of the proposed platform.

Conflict of interest

None declared.

Acknowledgments

Part of this work is supported by the COSARA-project: "Computer-based Surveillance and Alerting of Nosocomial Infections, Antimicrobial Resistance and Antibiotic Consumption in the Intensive Care Unit". Sofie Van Hoecke would like to thank the IWT (Institute for the Promotion of Innovation through Science and Technology in Flanders) for financial support through her Ph.D. grant. Filip De Turck acknowledges the F.W.O.-V. (Fund for Scientific Research-Flanders) for their support through a postdoctoral fellowship.

REFERENCES

[1] A. Morris, R. Gardner, Computer applications, in: J. Hall, G. Schmidt, L. Wood (Eds.), Principles of Critical Care, McGraw-Hill, New York, 1992, pp. 500–514.

[2] E.A. Hoste, G. Clermont, A. Kersten, R. Venkataraman, D.C. Angus, D. De Bacquer, J.A. Kellum, Clinical pulmonary infection score (CPIS) dynamics in polytrauma patients with ventilator-associated pneumonia, Crit. Care 10 (3) (2006) R73.

[3] P. Nystrom, The systemic inflammatory response syndrome: definitions and aetiology, J. Antimicrob. Chemother. 41 (suppl. A) (1998) P1–P7.

[4] A.C.K.-B. Amara, F.M. Andrade, R. Moreno, A. Artigas, F. Cantraine, J. Vincent, Use of the Sequential Organ Failure Assessment score as a severity score, Intensive Care Med. 31 (2005) 243–249.

[5] F. De Turck, J. Decruyenaere, P. Thysebaert, S. Van Hoecke, B. Volckaert, C. Danneels, K. Colpaert, G. De Moor, Design of a flexible platform for execution of medical decision support agents in the intensive care unit, Comput. Biol. Med. 37 (1) (2007) 97–112.

[6] S. Charbonnier, On line extraction of temporal episodes from icu high-frequency data: a visual support for signal interpretation, Comput. Methods Programs Biomed. 78 (2) (2005) 115–132.

[7] K.A. Thursky, M. Mahemoff, User-centered design techniques for a computerised antibiotic decision support system in an intensive care unit, Int. J. Med. Inform. 76 (10) (2007) 760–768.
[8] M. Bang, A. Larsson, E. Berglund, H. Eriksson, Distributed user interfaces for clinical ubiquitous computing applications, Int. J. Med. Inform. 74 (2005) 545–551.
[9] C. Hansom, B. Marshall, Artificial intelligence applications in the intensive care unit, Crit. Care Med. 29 (2001) 427–435.
[10] M.M. Levy, Computers in the intensive care unit, J. Crit. Care 19 (4) (2004) 199–200.
[11] Health informatics, Point-of-care medical device communication, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=36341(online).
[12] HL7, Health Level Seven, http://www.hl7.org/ (online).
[13] M. Fieschi, J. Dufour, P. Staccini, J. Gourvernet, O. Bouhaddou, Medical decision support systems: old dilemmas and new paradigms? Methods Inforrm. Med. 42 (2003) 190–198.
[14] R.A. Greenes, Clinical Decision Support: The Road Ahead, Academic Press, Inc., Orlando, FL, USA, 2006.
[15] H.-T. Chen, W.-C. Ma, D.-M. Liou, Design and implementation of a real-time clinical alerting system for intensive care unit, in: Proceedings of AMIA Symposium, 2002, pp. 131–135.
[16] K. Wakai, T. Kawamura, M. Endoh, M. Kojima, Y. Tomino, A. Tamakoshi, Y. Ohno, Y. Inaba, H. Sakai, A scoring system to predict renal outcome in IgA nephropathy: from a nationwide prospective study, Nephrol. Dial. Transplant. 21 (10) (2006) 2800–2808.
[17] C. Adrie, A. Cariou, B. Mourvillier, I. Laurent, H. Dabbane, F. Hantala, A. Rhaoui, M. Thuong, M. Monchi, Predicting survival with good neurological recovery at hospital admission after successful resuscitation of out-of-hospital cardiac arrest: the OHCA score, Eur. Heart J. 27 (23) (2006) 2840–2845.
[18] J. Chen, J. Chung, K.L. Wong, T. Fan, C.O. Pun, Early detection of pulmonary hypertension with heart sounds analysis pilot study, Stud. Health Technol. Inform. 122 (2006) 112–116.
[19] Sun Microsystems, Java Message Service Specification Version 1.1, http://java.sun.com/products/jms/, 2002.
[20] W.M. Tellis, K.P. Andriole, Integrating multiple clinical information systems using the java message service framework, J. Digital Imaging 17 (2) (2004) 80–86.
[21] S. Chen, P. Greenfield, Qos evaluation of JMS: an empirical approach, System Sciences, 10 pp, in: Proceedings of the 37th Annual Hawaii International Conference, 5–8 January, 2004.
[22] WS-Eventing, http://www.w3.org/Submission/WS-Eventing/, 2004.
[23] L. Macvittie, Make way for the ESB, Network Comput. 17 (5) (2006) 41–58.
[24] S. Desmet, B. Volckaert, S. Van Assche, D. Van Der Weken, B. Dhoedt, F. De Turck, Throughput evaluation of different enterprise service bus approaches, in: Proceedings of the 2007 International Conference on Software Engineering Research in Practice (SERP'07), 2007.
[25] J. de Keijzer, D. Tait, R. Goedman, JAIN: A new approach to services in communication networks, IEEE Commun. Mag. 38 (1) (2000) 94–99.
[26] Sun Microsystems, Java EE at a Glance, http://java.sun.com/javaee/ (online).
[27] The Apache Jakarta Project, Apache jmeter, http://jakarta.apache.org/jmeter/ (online).
[28] Open cloud, http://www.opencloud.com/.
[29] MuleSource Inc., Mule is the leading open source ESB and integration platform, http://mule.mulesource.org/ (online).
[30] B. Van Den Bossche, F. De Turck, B. Dhoedt, P. Demeester, Enabling Java-based VoIP backend platforms through JVM performance tuning, in: Proceedings of the First IEEE Workshop on VoIP Management and Security: VoIP MaSe co-located with IEEE NOMS 2006, 2006, pp. 41–47.
[31] Sun Microsystems, Tuning garbage collection with the 5.0 java[tm] virtual machine, http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html, 2003 (online).
[32] K. Verlaenen, B. De Win, W. Joosen, Towards simplified specification of policies in different domains, in: IEEE Conference on Integrated Network Management, 2007, pp. 20–29.
[33] Sun Microsystems, J2EE Connector Architecture 1.5, http://www.jcp.org/en/jsr/detail?id=112 (online).
[34] B. Van Den Bossche, F. De Turck, B. Dhoedt, P. Demeester, G. Maas, J. Moreels, B. Van Vlerken, T. Pollet, Evaluation of java-based middleware for service enabling platforms, in: Proceedings of the Symposium on Internet Services and Enabling Technologies (ISET), IEEE Globecom 2006, 2006.