Notification System

Ayemi Musa, Ankan Mookherjee,Gregory Wright
Rochester Institute of Technology

ABSTRACT

Many student information systems like RIT SIS perform course cataloging and student enrolment, and automatically generate weekly time schedule, but unable to synchronize with calendar application. To configure calender reminders, students must manually enter their time table in a calendar application like Microsoft Outlook, Google Calendar, etc. We present a lightweight notification system on the cloud that provide class timetable reminder service to interested subscribers. Based on distributed, loosely coupled services/components, the system is designed to serve a large subscriber base; therefore, it is imperative that the system is able to tolerate failure and guarantee delivery of notification within minimum jitter.

1. OVERVIEW

We present a distributed notification system built on loosely coupled services deployed across multiple instances of Amazon Elastic Compute Cloud (Amazon EC2) and MuleSoft CloudHub. This application would allow multiple users to subscribe to Class time table they wish to be reminded of. Once they subscribe, the notification engine would send out message alerts at a specified time before the beginning of the desired class. Also, it permits subscribers to opt out of their subscription and further notifications will not be sent out.

2. DESIGN CONSIDERATIONS

In build this system, a lot of considerations went into the design which influence the system architecture in several dimensions. The main decisions concern quality attributes - observable properties that are desired from the user's perspective. The system must be fault tolerant, highly available, and timely as delivery of notification must be guaranteed within stringent time limits. Given the total enrollment at RIT of approximately 20000, the system is expected to process an increasing number of transactions concurrently therefore scalability was also an underpinning factor.

3. ARCHITECTURE

The system design is based on a distributed multi-tiered and service oriented architecture (SOA).

3.1 Client Tier

The client tier consists of a light weight web frontend running on laptops and mobile devices to access services in the business tier.

3.2 Business Tier

This tier consists of SOAP web service endpoints which implements the subscription and notification services. The subscription service receives subscription requests from the client side and places the request on a queue for processing, and the notification services delivers message alert to the client as events occur. We implemented a system that mimics SIS (mini SIS) as another web endpoint that provides course cataloging, student enrollment and information service.

3.3 Enterprise Information System Tier (EIS)

The EIS tier consists of a database system as the main source of data.

The architecture is supported by a lightweight enterprise service bus (ESB) middleware to provide application integration, message routing capability, publish/subscribe functionality, load balancing, and communication protocols abstraction. In addition, the ESB provides failure masking by implementing retries and dead letter queue (DLQ) in the event of service unavailability.

4. IMPLEMENTATION

4.1 Subscription Request

This is the client-facing interface that receive incoming subscription request via http protocol. An inbound-endpoint based on http transport is configured in the ESB to intercept incoming messages on the specified port and URL path. The payload is placed on a message queue to guarantee that it will eventually be processed, and a friendly response is returned to the client. The rationale for using a message queue is to ensure that no message is lost in the event of any failure, and to decouple the client from the front-end subscription web service.

4.2 Subscription Service

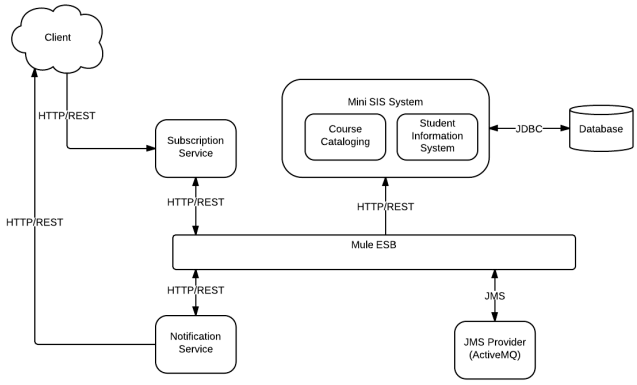


Figure 1: High-level Architecture

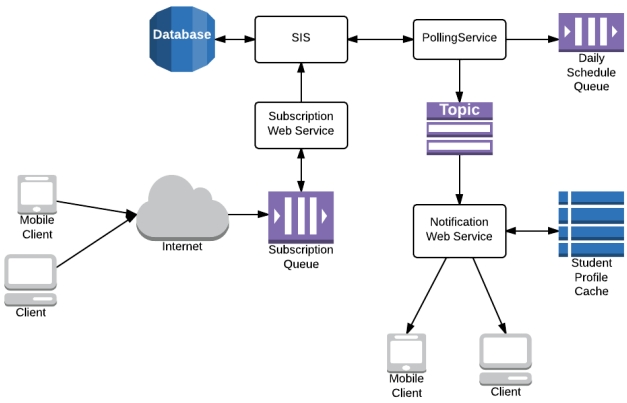


Figure 2: Implementation View

The function of this service is to listen for incoming message in the queue and forward it to the SIS service. The subscription service is implemented as a Mule flow.

4.3 SIS

SIS is a SOAP web service endpoint that exports various public operations to fulfill the functions of student information service.

- `getClassEnrollement:` This operation returns the list student enrolled for a particular class schedule.
- `getClassSubscription:` This operation returns the list student subscribed for a particular class schedule.
- `getStudentProfile:` This operation returns a single student profile.
- `getStudentSubscription:` This operation returns the list of a students class subscription.
- `getStudentEnrollement:` This operation returns the list of a students class enrollment.
- `getClassScheduleForToday:` This operation returns the list of class schedule for the current day.

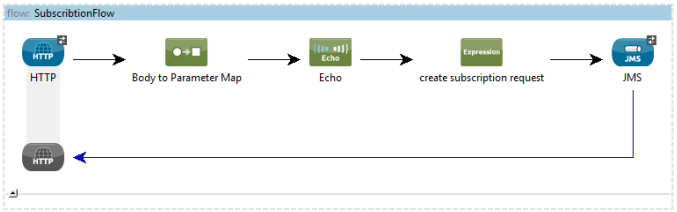


Figure 3: Subscription Request

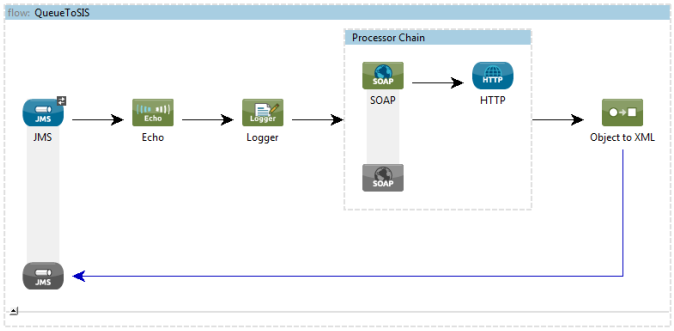


Figure 4: Subscription Service

- `subscribe:` This operation receives a subscription request and persist in the enrollment table.
- `unsubscribe:` This operation removes a student from the subscription record.

4.4 Polling Service

Several implementation options have been considered for polling mechanism: Java message beans, Quartz connector, Java component. Each of these options could serve the purpose but we decided to implement polling using the Poll [1] scope from the Mule ESB. The Poll scope is a component which periodically polls an embedded message receiver for new messages. Specifically, we configured a SOAP component (JAX-WS client) within the Poll processing block to retrieve daily class schedule information from the SIS service at regular interval (24 hours). Two levels of polling exists:

4.4.1 24 Hour Polling:

This is the main poll that triggers a JAX-WS client which consumes the `getClassScheduleForToday` operation of the SIS service to retrieve the daily class schedules. The daily class schedule is the list of classes that are due each day, and at every poll the class schedules are placed in a queue for further processing. For the purpose of efficiency, the list of class schedules are processed by a Java component which organizes the class schedules according to start time. All classes beginning at each round of an hour are grouped together in one list and put in a map identified by the start time key.

4.4.2 Just-In-Time (JIT):

[1] www.mulesoft.org/documentation/display/current/Scopes

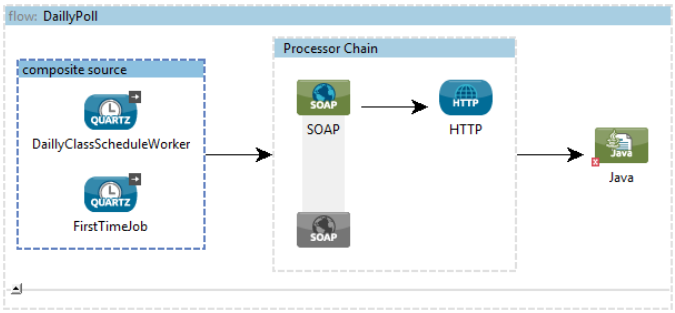


Figure 5: JAX-WS Polling Service

This polling is implemented using the Quartz[2] component which also provides support for scheduling events and for triggering new events. We used an outbound quartz endpoint to schedule periodic calls to the Java component method that checks if the next class hour will be due in the next 5 minutes and place the schedules on a special JMS topic which will get delivered to the appropriate endpoint to generate and distribute notifications.

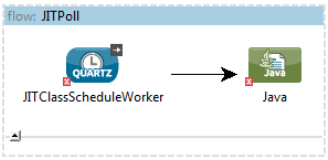


Figure 6: Class Time Polling Service

4.5 Notification Service

The notification service receives class schedules that are due for notification via a JMS topic, and its job is to identify those students who are subscribed for that event. The service maintains a cache for optimum performance to look up subscribers (students) and determine their preferred communication medium (email or SMS) and asynchronously send notifications. The Notification service uses the SMTP component to for sending email messages over SMTP using the `javax.mail.API`. The component supports CC/BCC/ReplyTo addresses, attachments and custom Header properties. It also provides support for `javax.mail.Message` transformation. The SMTP component is simply a transport layer and it depends on an email server for the heavy-lifting emailing functions. Initially, we tried using Amazon Simple Email Service (SES) as the SMTP endpoint but unsuccessful because of Amazon's restriction policies on the emailing platform - it requires domain validation and other miscellaneous policies that prevents us from using it. At design-time we used the open source Java Email Server (JES) to test the email functionality of the system, but now our notification system uses Elastic Email cloud based SMTP service.

4.6 Distributed Considerations

We use Amazon Elastic Compute Cloud (Amazon EC2) which provides resizable cloud computing capacity and system-level services like security, availability, and low-level load balancing. For the most part we leverage the services offered out-of-the-box by the cloud providers to address distributed concerns.

[2]www.mulesoft.org/documentation/display/current/Quartz+Transport+Reference

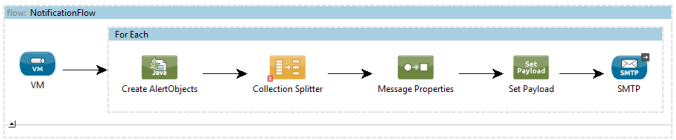


Figure 7: Notification Service

4.6.1 *Heterogeneity*

Our system is design with open standards in mind to allow various type of clients devices to access the services. While the backbone of the system platform is based on an Enterprise Service Bus that is capable of masking heterogeneity, we opted for HTTP protocol which is a widely adopted standard for web communications that is supported by most or all internet browsers. In the future we plan to utilize the Mule ESB connectors to allow various native protocols including low level communication primitives to communicate directly with our system and in a transparent manner.

4.6.2 *Publish-subscribe*

Publish-subscribe is a communication pattern that decouples senders and receivers in space and time so that senders of messages, called publishers, do not need to know any specific receiver to send message directly to. Similarly, receivers express interest in one or more types of message, and only receive messages that are of interest, without knowledge of what, if any, senders there are. We implement this messaging pattern using Apache ActiveMQ in order to provide greater scalability for our notification system. However, our implementation is sort of a hybrid type where the subscriptions are manage in a database rather than in the pub/sub middleware. This concession is to allow us implement time polling in order to trigger time based message publishing.

4.6.3 *Distributed Coordination*

Distributed coordination is handled in two folds: (a) The underlying Amazon Web Service platform and (b), the underlying integration and routing mechanism of the Mule Enterprise Service Bus which comes out-of-the box. Detailed description of how these works is out of the scope of this paper.

4.6.4 *Fault tolerance*

Amazon EC2 offers a fault tolerant architecture, in addition to that we use a message queue to receive incoming request in order to guarantee deliver even in the event of unexpected fault. Once the message is received in the queue, even if processing fails, the message will remain in queue and when the service restarts, a retry will be initiated until successful. When the request times-out, the request will still be preserved by placing it in a dead letter queue (DLQ).

4.6.5 *Availability*

The degree to which our system is in an operable and functioning condition at any point in time defines the availability of the system. Amazon EC2 platform is highly ex-

tensible and replicated according to availability zones, and Amazon commits to a service level agreement of over 99.9% uptime. In addition, we implement a caching strategy to increase performance and maximize availability of the system even when the database is unavailable. The caching strategy also optimizes performance by reducing the amount of round-trips.

4.6.6 *Cloud Deployment*

Our deployment strategy is strictly cloud-base involving multiple server instances for fail-over and redundancy. We rely on Amazon location-aware replication for each server instance and also load balancing mechanism. We have an instance for the SIS system and one instance for the database server hosted on the Amazon EC2 [1]. We do not require the database instance to have much load balancing and replication because it is not a major point of failure since we are relying mostly on caching for polling, and message queue to retain subscription requests before being persisted in the database. Therefore we expect minimal database hits. In addition, we deployed our integration code stack on an the production tier of the Mule CloudHub[3] platform. The Mule application is the backbone of our distributed system, which provides the service orchestration and message routing for our application and the hub of our cloud-to-cloud integration and service oriented architecture.

4.6.7 *Security*

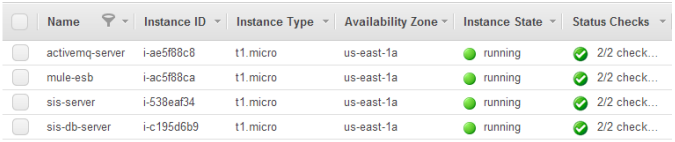
We implement a typical two-level security model: (a) instance level, and (b) OS level security. Amazon EC2 secures instances by means of a Security Group configuration, which acts as a firewall setting that controls traffic to and from the virtual machine instances. We modify the security group settings to control inbound and outbound network access our instances and designated ports. Furthermore, we reinforce our security by configuring OS level firewall rules.

4.6.8 *Scalability: Elastic Load Balancing*

Elastic Load balancing feature automatically distributes incoming application traffic across multiple Amazon EC2 instances. This enables us to achieve fault tolerance and equalization of workload across virtual machine and application instances in response to incoming application traffic.

4.6.9 *Replication*

Amazon operates highly available state-of-the-art data centers, and enables data and resource sharding across multiple data centers. This allows our application to be highly available at all times.



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks
activemq-server	i-ae5f88c8	t1.micro	us-east-1a	running	2/2 check...
mule-esb	i-ac5f88ca	t1.micro	us-east-1a	running	2/2 check...
sis-server	i-538eaf34	t1.micro	us-east-1a	running	2/2 check...
sis-db-server	i-c195d6b9	t1.micro	us-east-1a	running	2/2 check...

Figure 8: AWS EC2 Instances

5. LESSONS LEARNED

Based on the fact that the initial architectural design is at a conceptual level, the design changes are not readily obvious. Whereas the design is largely the same, it can be noted that the subscription part and the polling mechanism are handled outside the pubsub middleware (JMS provider). Client subscription requests are persisted in a database through the SIS web service and the polling mechanism is handled by the underlying ESB platform. This design decision was necessitated by the constraints encountered during prototyping of the architecturally significant requirements of the system. The intent of the initial design was to have client request subscribed against a class topic identified by the class number of interest, and the JMS provider should automatically self-publish to the class topic and trigger notification to the various subscribers based on the occurrence of time (i.e. when the class time starts). However, that wasn't possible and we investigated and found out that one of the major impedances was the fact that JMS implements a non-polling protocol, and as such it is impossible to realize the original design, therefore we had to resort to the new design. Additionally, we have been confronted with global clock fallacy issues as we come to the realization that distributed systems must be designed to deal with the issue. While our application works pretty fine on our local machine by deliverying notification at the correct time, things changed as soon as we deployed the notification system on the cloud, and it turns out that notification events are triggered at the wrong timing. It wasn't immediately obvious that this was a result of the global clock issue, until after much troubleshooting.

6. CURRENT STATUS AND FUTURE WORK

Currently we have implemented the routing mechanism for receiving and placing client request in the message queue, and returning an acknowledgement to the client. Also, we have implemented the routing of client subscription request received in the queue to be persisted in the database through the SIS `subscribe` web service operation. We have implemented all other web services required for our application including the `subscribe`, `getClassScheduleForToday`, `getStudentProfile`, `getClassEnrollment`, `getClassSubscription`, `getStudentEnrollment`, and `getStudentSubscription` operations. The outstanding operation yet to be implemented is `unsubscribe` to allow clients opt out from their current subscription. Also pending is the implementation of an efficient caching strategy for optimum performance. We complete our end-to-end subscribe-notification cycle with the integration of a cloud SMTP service for delivering email notifications. In the future we will also include SMS communication medium. Furthermore, we will address the issue of global clock to allow the system in the future to take cognizance of the subscriber's timezone when a subscription is made, so that notifications will be delivered based on the originating timezone instead of the instance availability zone.

In summary, we have delivered a distributed notification system based on cloud-to-cloud service integration.

6.1 Major Change

Polling mechanism: In the previous phase we implemented a step-wise polling mechanism using various pollers for class schedules: A 24-hour poller polls the SIS `getClassScheduleForToday` operation on a daily basis to obtain daily class schedule information and place on

a 24-hour queue. A 1-hour poller polls the 24-hour queue for classes that are due within the hour, and finally a just-in-time poller which polls the 1-hour queue every 5 minutes for classes that are due and place on the JMS topic for further processing. However, this mechanism has been replaced with a more efficient algorithm or approach to processing the class schedule. Basically, the rational for the new approach is based on the fact that there are only limited class periods in a day (maximum of 14) - i.e. one class every hour from 8 am to 9 pm. Therefore, instead of polling every individual class schedule, the class schedules are organised in according to their start-time in a `HashMap` data structure, and the poller only polls on the class period start times and retrieves the list of classes by the start time key. The new approach is described in details in section 4.4 the the detailed implementation view in figure reflects the new changes.

Cloud Deployment: Previously, we have our application deployed on the local machine, but now we have created provisioned EC2 server instances to host our SIS system and database.

7. RELATED WORK

Other reminder applications currently exist that are similar to our application. Google Calendar is a free application released in April 2006 [2]. The system works with a calendar based application. The application synchs the calendar with its notification through email or short message system. The Google Calandar system is more than a reminder system, with the ability to use Google Sync to interact with other Google applications. The Google Calendar application is developed with Ajax and offers a large amount of user customization. Another reminder application is Due for Apple devices [4]. This application is a simpler system that is built for the IOS systems. The application allows the user to input specific reminders for single or multiple times. The Due system doesn't have a calendar of its own, but rather uses other calendar systems of the device. This does seem to be a limitation that the Google Calendar application does not have. These reminder applications are totally user driven. The user inputs their own reminders and their own calendars into the application. Our application is different in that the user is applying to previously determined events and receiving notification.

8. REFERENCES

[1] A. Inc. Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2/.

[2] G. Inc. Google apps for business. http://www.google.com/intx/en/enterprise/apps/business/products.html#calendar.

[3] MuleSoft. Mulesoft cloudhub. http://www.mulesoft.com/cloudhub/ipaas-cloud-based-integration-demand.

[4] Phocus. Jot down a task and set up a reminder really, really fast? http://www.dueapp.com/.