LINKEDIN COMMUNICATION ARCHITECTURE

Ruslan Belkin, Sean Dawson

TS-5234

Java

Sun microsystems

Learn how we at LinkedIn built and evolved scalable communication platform for the world's largest professional network

GOAL

Agenda

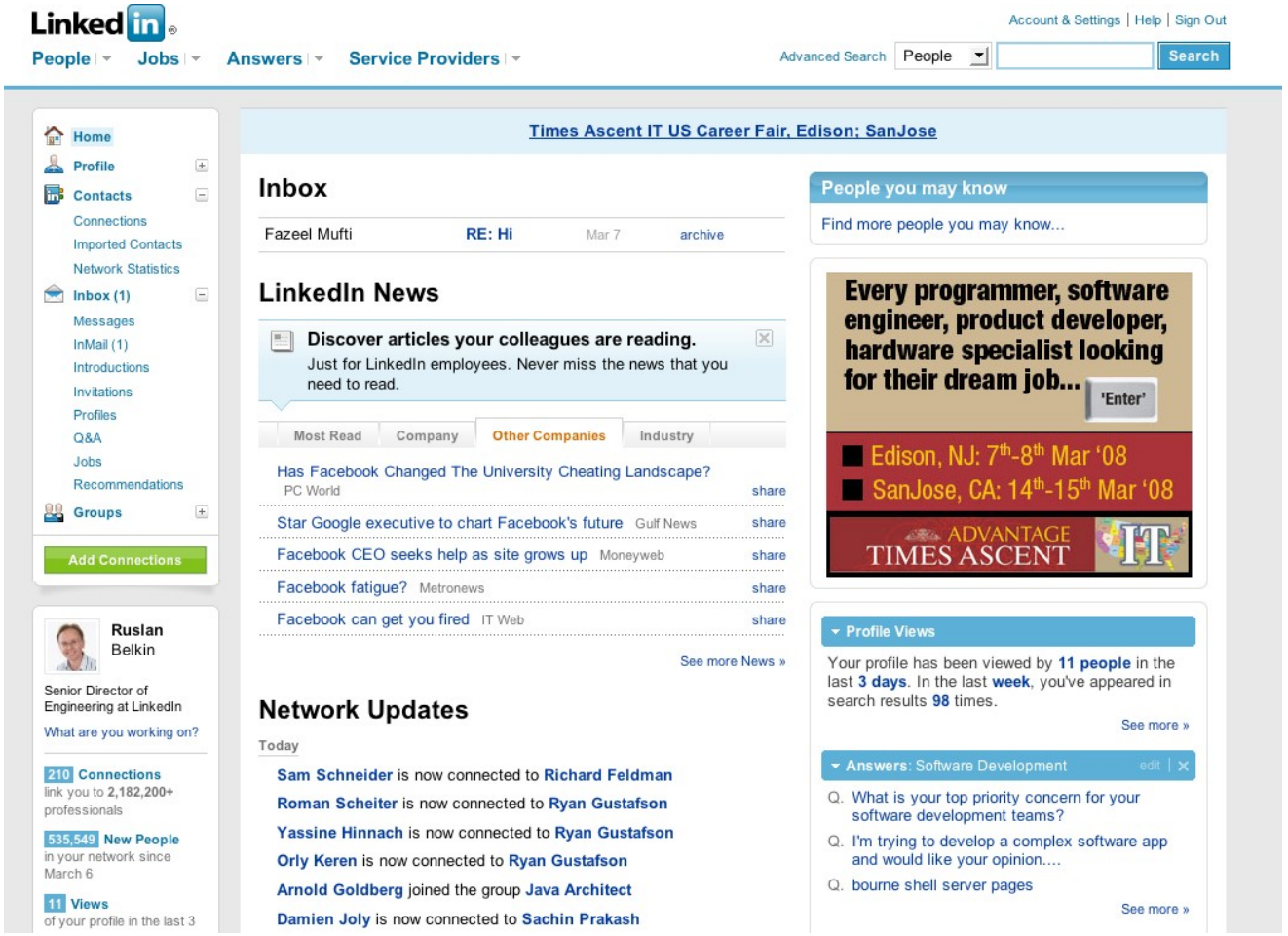
- Why are we doing this talk
- LinkedIn Communication Platform at a glance
  - Evolution of LinkedIn Communication System
  - Evolution of the Network Updates System
- Scaling the system: from 0 to 22M members
- Q&A

Why are we doing this talk?

- Share our experience in building the world-largest professional network in Java™
- Describe the evolution of the communication platform
- Share lessons we learned so you could benefit from our successes, mistakes and experience

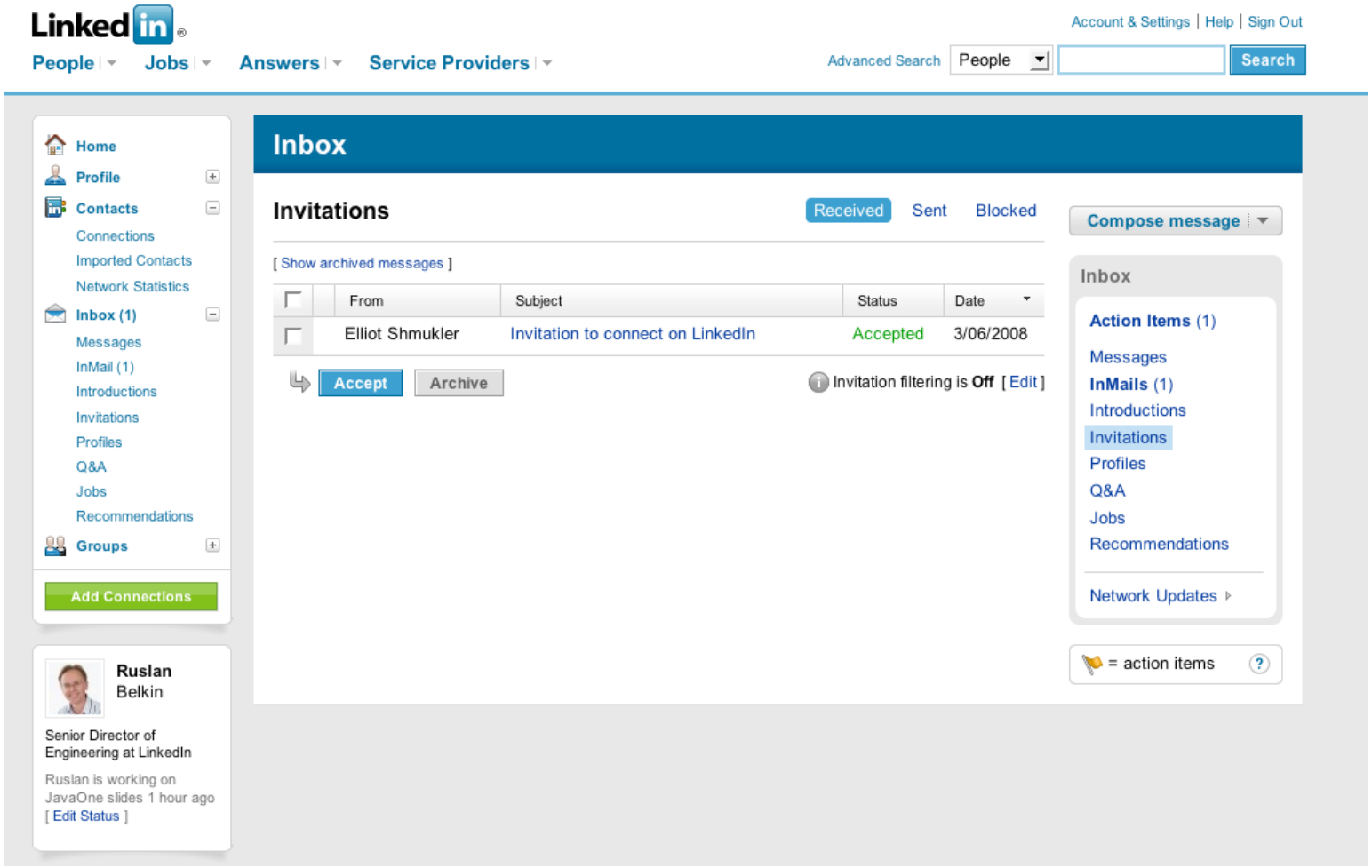
LinkedIn Communication Platform

Quick Tour



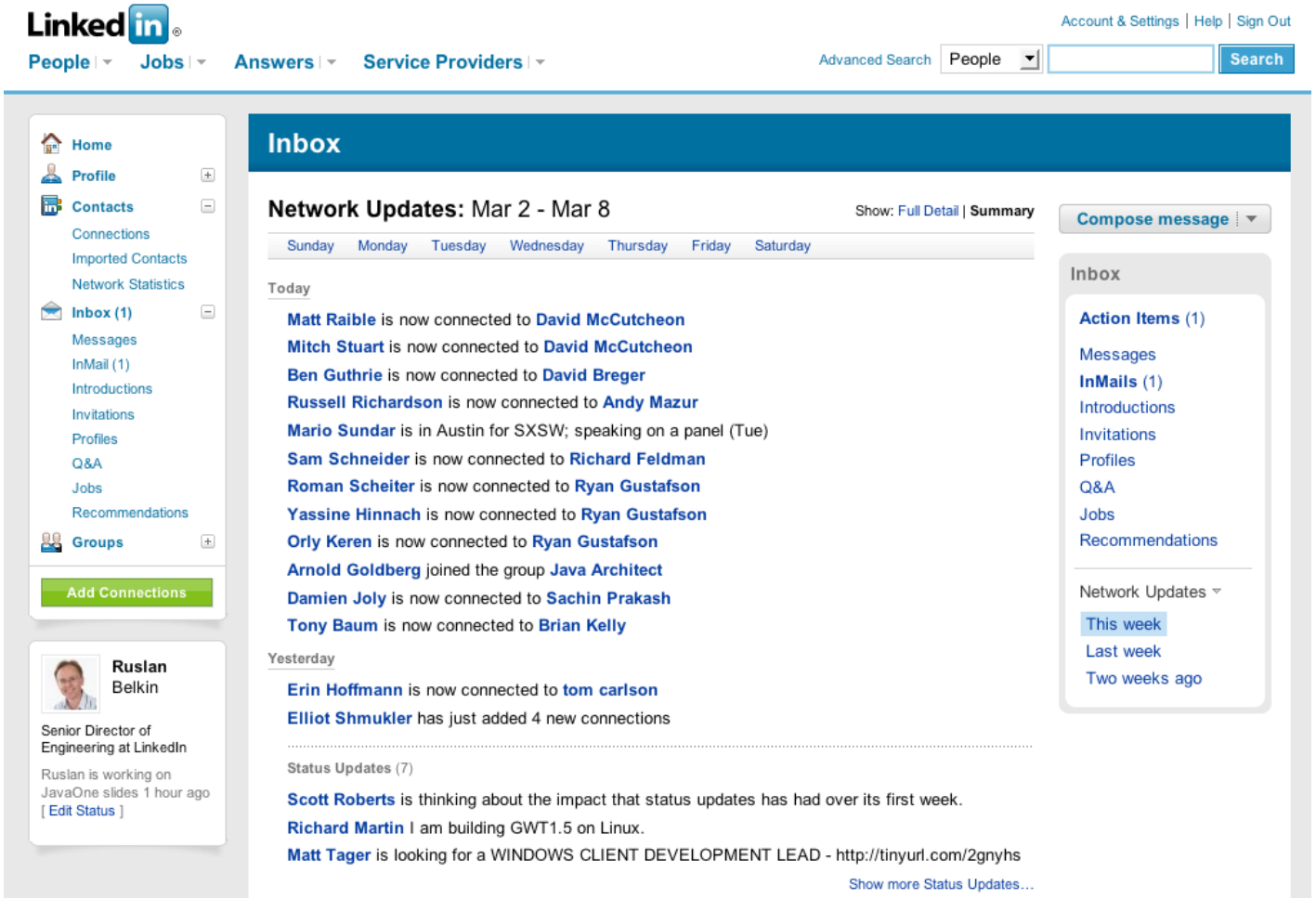
LinkedIn Communication Platform

Quick Tour



LinkedIn Communication Platform

Quick Tour



LinkedIn Communication Platform

The Numbers

- 22M members
- 130M connections
- 2M email messages per day
- 250K invitations per day

LinkedIn Communication Platform

The Setup

- Sun™ x86 platform and Sparc production hardware running Solaris™ Operating System
- 100% Java programming language
- Tomcat and Jetty as application servers
- Oracle and MySQL as DBs
- ActiveMQ for JMS
- Lucene as a foundation for search
- Spring as a glue
- Mac for development

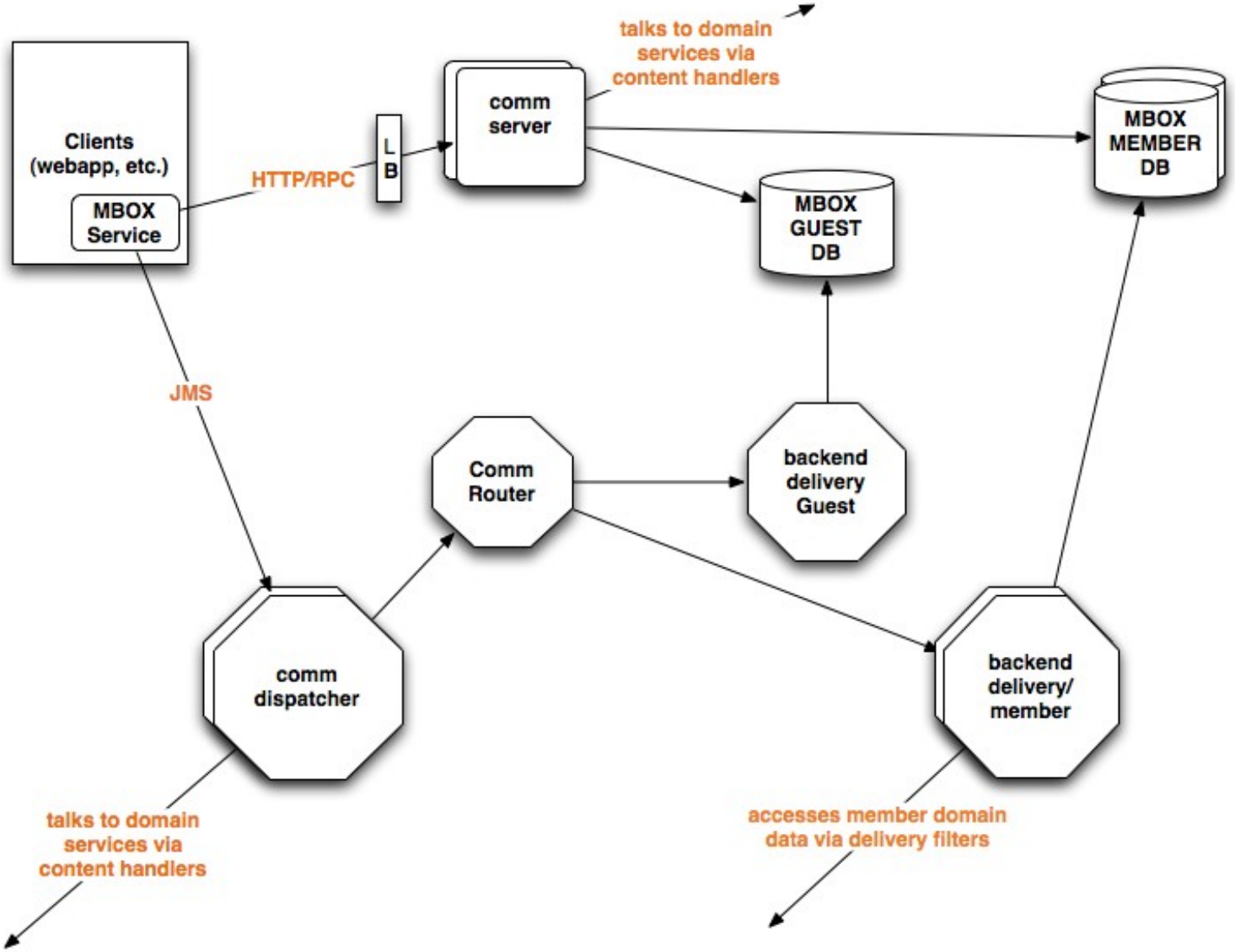
LinkedIn Communication Platform

- The Communication Service
  - Permanent message storage
  - InBox messages
  - Emails
  - Batching, delayed delivery
  - Bounce, cancellation
  - Actionable content
  - Rich email content
- The network updates service
  - Short-lived notifications (events)
  - Distribution across various affiliations and groups
  - Time decay
  - Events grouping and prioritization

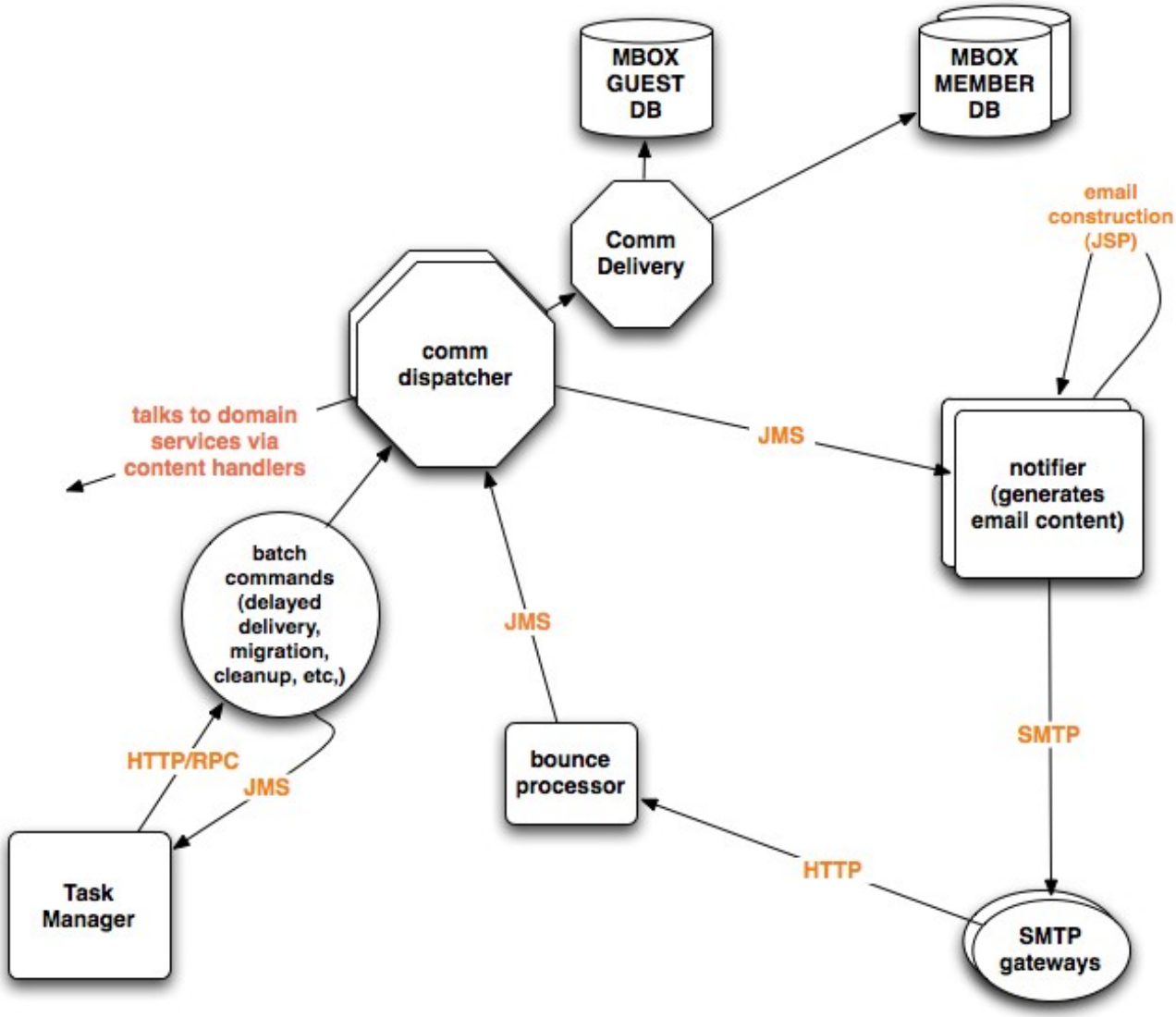
The Communication Service

- How is it different:
  - Workflow oriented
  - Messages reference other objects in the system
  - Incorporates email delivery
  - Batching of messages
  - Message cancellation
  - Delayed delivery, customer service review queues, abuse controls
  - Supports reminders and bounce notifications to users
- Has undergone continuous improvements throughout life of LinkedIn

Message Creation



Message Delivery



The Communication Service

- Message Creation
  - Clients post messages via asynchronous Java Communications API using JMS
  - Messages then are routed via routing service to the appropriate mailbox or directly for email processing
  - Multiple member or guest databases are supported
- Message Delivery
  - Message delivery is triggered by clients or by scheduled processes
  - Delivery actions are asynchronous
  - Messages can be batched for delivery into a single email message
  - Message content is processed through the JavaServer Page™ (JSP™) technology for pretty formatting
  - The scheduler can take into account the time, delivery preferences, system load
  - Bounced messages are processed and redelivered if needed
  - Reminder system works the same way as message delivery system

The Communication Service

- SOA architecture
- Wireable components build around LinkedIn Spring extensions
- Spring HTTP-RPC
- Heavy use of JMS and asynchronous flows

The Communication Service

Failure Recovery

- Possible failures:
  - Messages can bounce
  - Messages can get lost:
    - Database problems
    - Bugs in the code
    - Bugs in the content processing of emails
    - Various services may become unavailable
- Avoiding the downtime

The Communication Service

How do we scale it?

- Functional partitioning:
  - sent, received, archived, etc.
- Class partitioning:
  - Member mailboxes, guest mailboxes, corporate mailboxes
- Range partitioning:
  - Member ID range
  - Email lexicographical range
- Asynchronous flows

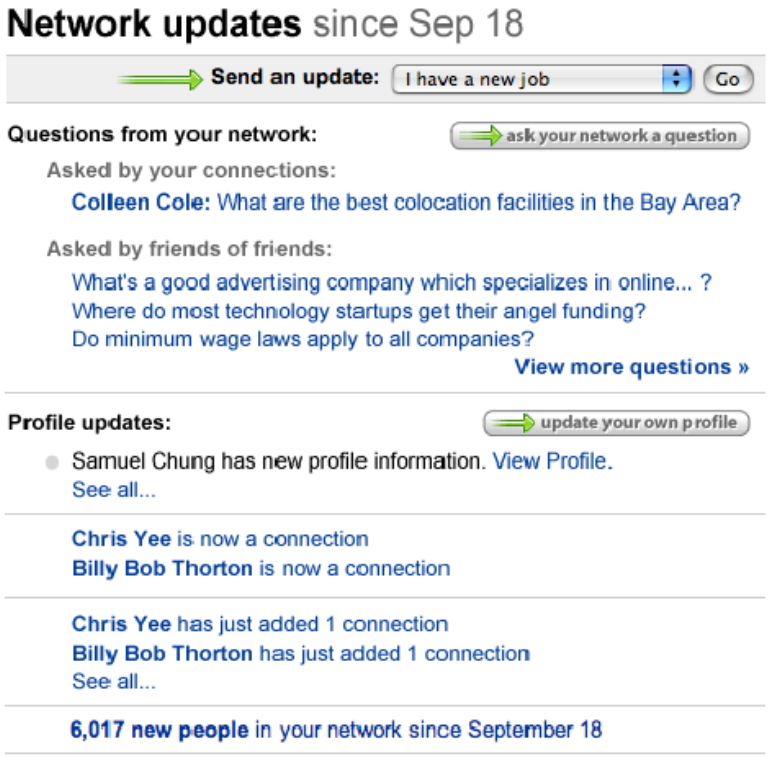
Network Updates Service

- What is your network up to?
- The goal is to have a flexible service for distributing many types of *short-lived* updates
- Availability across a number of clients (web apps, RSS, API, LinkedIn Mobile, third-party…)

Network Updates Service

Motivation

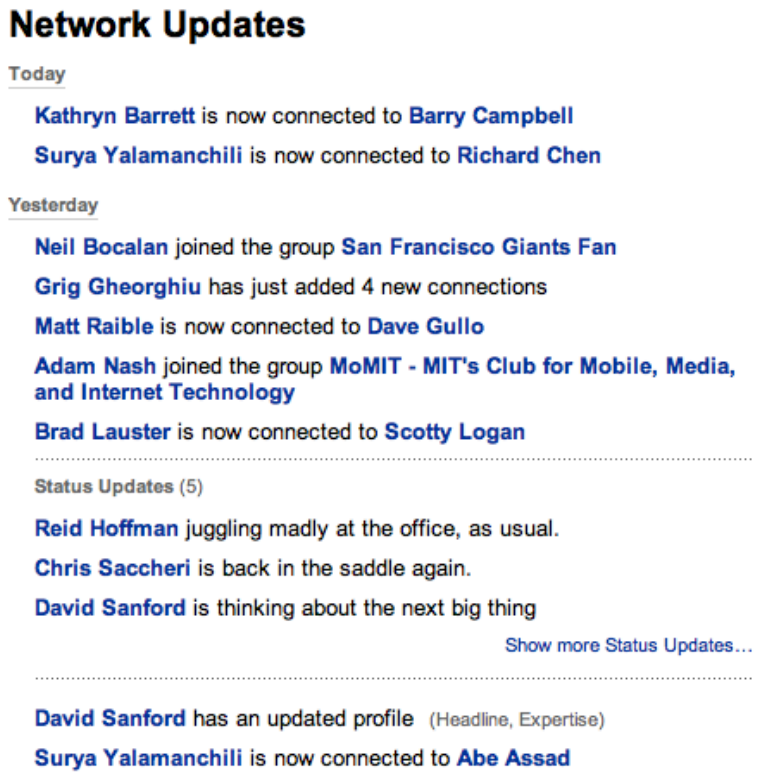
- Homepage circa 2007
- Poor UI
  - Cluttered
  - Where does new content go?
- Poor Backend Integration
  - Many different service calls
  - Takes a long time to gather all of the data



Network Updates Service

Motivation

- Homepage circa 2008
- Clean UI
  - Eliminates contention for homepage real estate
- Clean Backend
  - Single call to fetch updates
  - Consistent update format



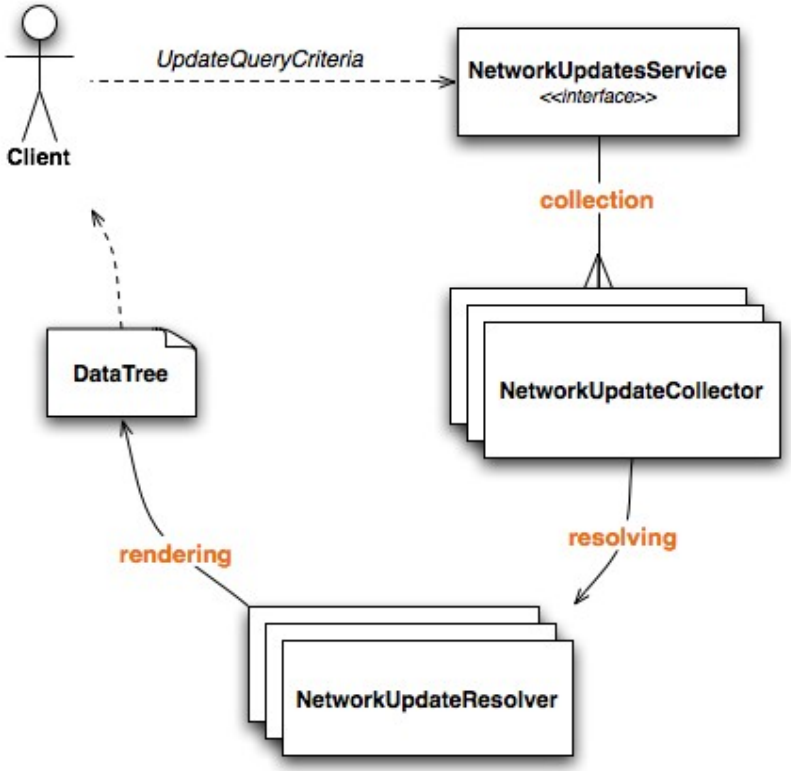
Network Updates Service

Iteration 1

- Move existing homepage logic into a remote service, refactor homepage to use the new service
- Advantages
  - No user impact while API is being finalized
  - Improve performance by fetching updates in parallel
  - Reduce complexity of the web app
  - Updates become easily accessible to other clients

Network Updates Service

Iteration 1 - API



Network Updates Service

Iteration 1 - API

- Pull-based architecture
- Collectors
  - Responsible for gathering data
  - Parallel collection to improve performance
- Resolvers
  - Fetch state, batch lookup queries, etc…
  - Use EHCache to cache global data (e.g., member info)
- Rendering
  - Transform each object into its XML representation

Network Updates Service

Iteration 1 - Example

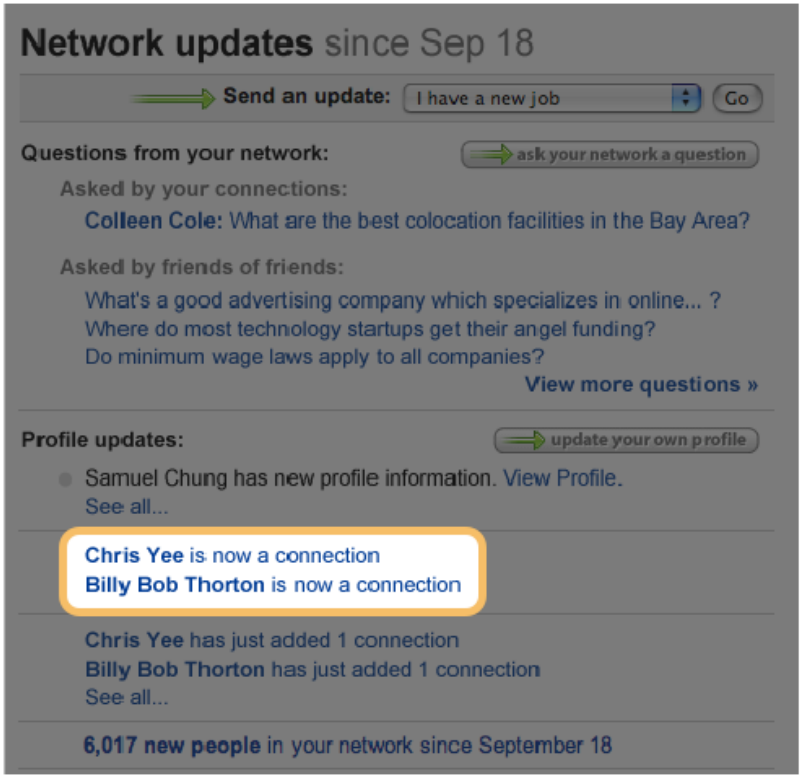
```
UpdateQueryCriteria query =
   UpdateQueryCriteria.createDefaultCriteria()
        .setMemberID(2)
        .setRequestedTypes(NetworkUpdateType.CONNECTION)
        .setCutoffDate(…)
        .setMaxNumberOfUpdates(10);

MyNetworkUpdatesService service =
   getService(MyNetworkUpdatesService.class);

DataTree update =
   service.getNetworkUpdatesSumary(query);
```

Network Updates Service

Iteration 1 - Example




```
<updates>
  <NCON>
    <connection>
      <id>2</id>

    <firstName>Chris</firstNa
    me>

    <lastName>Yee</lastName>
    </connection>
  </NCON>
</updates>
```

Network Updates Service

Iteration 1

- Lessons learned:
  - Centralizing updates into a single service leaves a single point of failure
  - Be prepared to spend time tuning the HttpConnectionManager (timeouts, max connections)
  - While the system was stabilizing, it was affecting all users; should have rolled the new service out to a small subset!
  - Don't use "Least Frequently Used" (LFU) in a large EHCache—very bad performance!

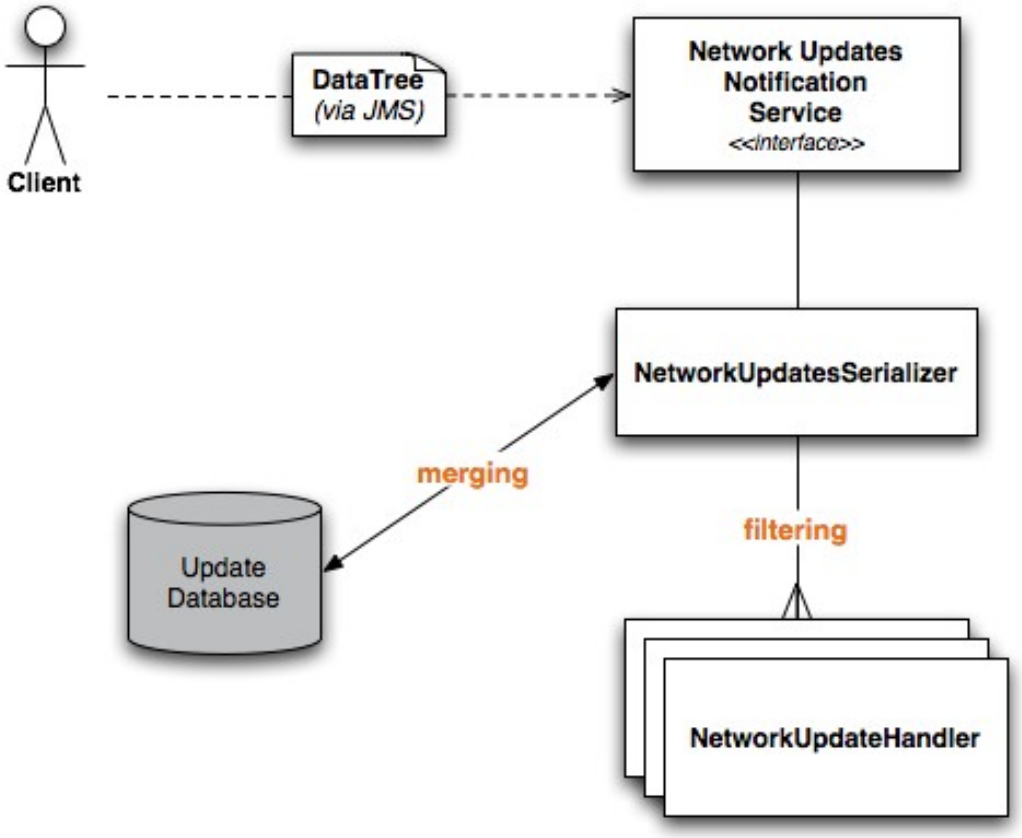
Network Updates Service

Iteration 2

- Hollywood Principle: *"Don't call me, I'll call you"*
- Push update when an event occurs
- Reading is much quicker since we don't have to search for the data!
- Tradeoffs
  - Distributed updates may never be read
  - More storage space needed

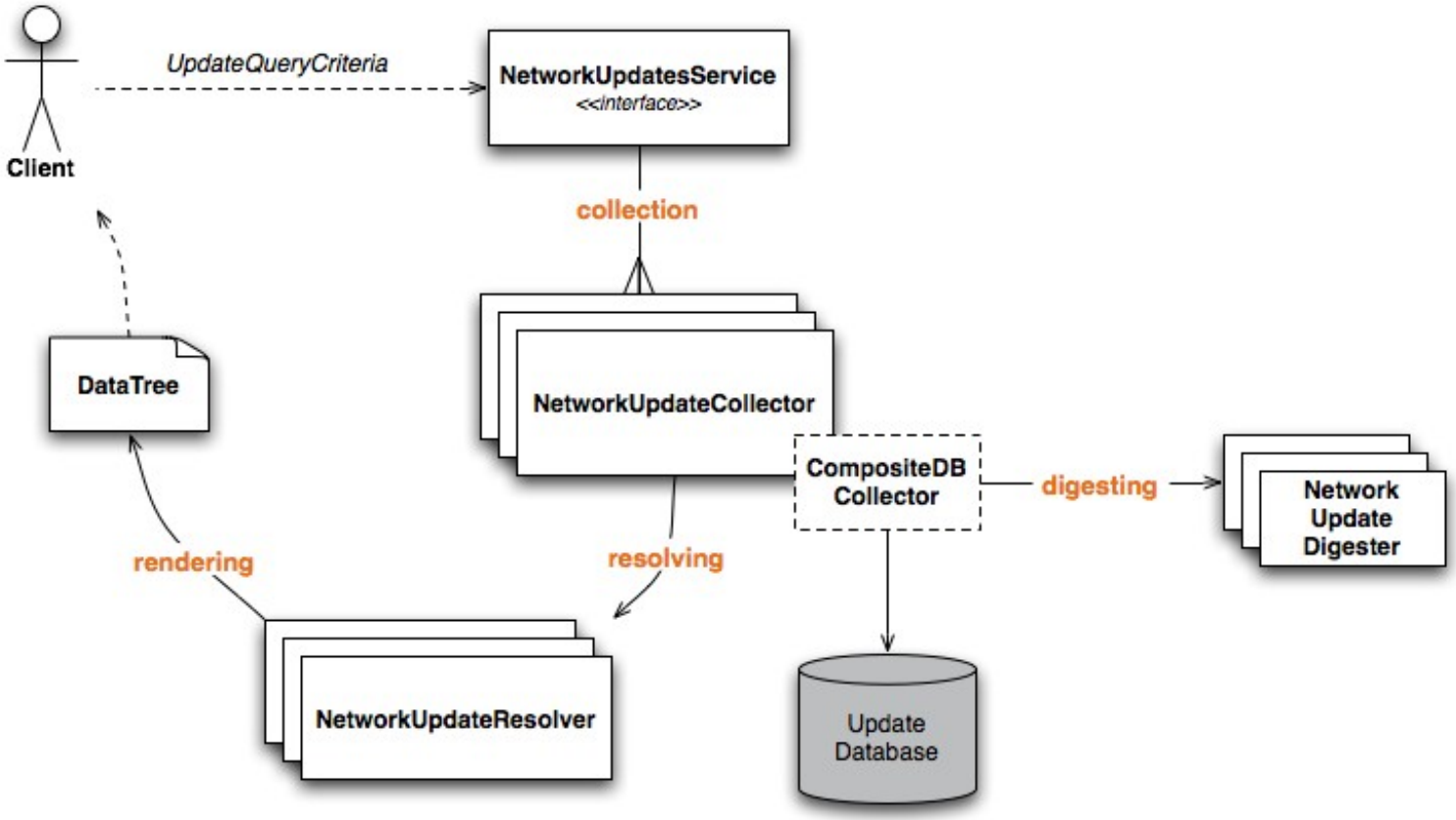
Network Updates Service

Iteration 2 - Pushing Events



Network Updates Service

Iteration 2 - Reading Updates



Network Updates Service

Iteration 2

- Pushing Updates
  - Updates are delivered via JMS
  - Aggregate data stored in 1 CLOB column for each target user
  - Incoming updates are merged into the aggregate structure using optimistic locking to avoid lock contention
- Reading Updates
  - Add a new collector that reads from the Update Database
  - Use Digesters to perform arbitrary transformations on the stream of updates (e.g, collapse 10 updates from a user into 1)

Network Updates Service

Iteration 2

- Lessons learned:
  - Underestimated the volume of updates to be processed
  - CLOB block size was set to 8k, leading to a lot of wasted space (which isn't reclaimed!)
  - Real-time monitoring/configuration with Java Management Extension (JMX™) specification was extremely helpful

Network Updates Service

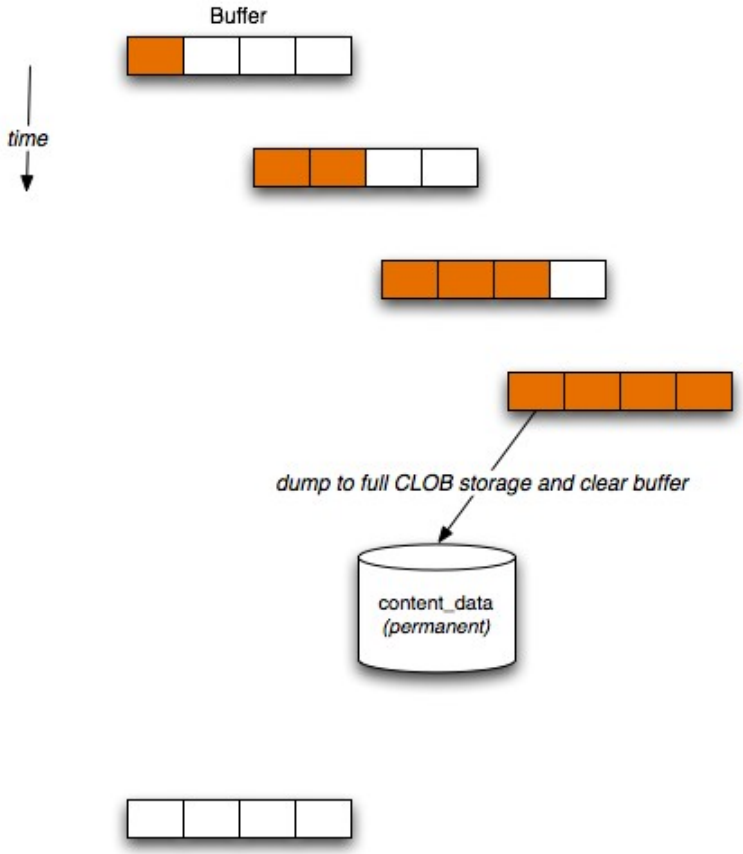
Iteration 3

- Updating a CLOB is expensive
- Goal: Minimize the number of CLOB updates
  - Use an overflow buffer
  - Reduce the size of the updates

Network Updates Service

Iteration 3 - Overflow Buffer

- Add VARCHAR(4000) column that acts as a *buffer*
- When the buffer is full, dump it to the CLOB and reset
- Avoids over 90% of CLOB updates (depending on type), while still retaining the flexibility for more storage



Scaling the system

- What you learn as you scale:
  - A single database does not work
  - Referential integrity will not be possible
  - Cost becomes a factor: databases, hardware, licenses, storage, power
  - Any data loss is a problem
  - Data warehousing and analytics becomes a problem
  - Your system becomes a target for spamming exploits, data scraping, etc.
- What to do:
  - Partition everything:
    - by user groups
    - by domain
    - by function
  - Caching is good even when it's only modestly effective
  - Give up on 100% data integrity
  - Build for asynchronous flows
  - Build with reporting in mind
  - Expect your system to fail at any point
  - Never underestimate growth trajectory

LinkedIn Communication Architecture

- Build with scalability in mind - never know when your business will take off
- Expect to do a lot of architecture and code refactoring as you learn and appreciate growth challenges

THANK YOU

LinkedIn Communication Architecture

Ruslan Belkin (http://www.linkedin.com/in/rbelkin)
Sean Dawson (http://www.linkedin.com/in/seandawson)

We are hiring!

The Communication Service

LinkedIn Spring Extensions

- Automatic context instantiation from multiple spring files
- LinkedIn Spring Components
- Property expansion
- Automatic termination handling
- Support for Builder Pattern
- Custom property editors:
  - Timespan (30s, 4h34m, etc.)
  - Memory Size, etc.

The Communication Service

LinkedIn Spring Extensions

```
Comm-server/
    cmpt/
        components/
                ccsServiceExporter.spring
                comm.spring
                jmx.spring
                comm-server.properties
                        corpMboxServiceExporter.spring
                main.spring
                comm-server.spring
                memberMboxServiceExporter.spring
                comm.properties
                guestMboxServiceExporter.spring
        build.xml
     impl/
        …
```

The Communication Service

LinkedIn Spring Extensions

```
…
<bean id="resolver”
class="com.linkedin.comm.pub.impl.MessageAddressResolver">
    <lin:config>
        <property name="resolverDB" ref="resolverDB"/>
        <property name="eos" ref="eos"/>
        <property name="els" ref="eos"/>
        <property name="memberAccessor"
ref="coreMemberAccessor"/>
    </lin:config>
</bean>
…
```

The Communication Service

LinkedIn Spring Extensions (Builder)

```
private final MessageAddressResolverDB _resolverDB;
…
MessageAddressResolver(Config config) { … }
…
public static class Config    {
    private MessageAddressResolverDB _resolverDB;

    public MessageAddressResolverDB getResolverDB() {
        return ConfigHelper.getRequired(_resolverDB);
    }
    public void setResolverDB(MessageAddressResolverDB
                                    resolverDB) {
                _resolverDB = resolverDB;
    }
}/*Config*/
```

The Communication Service

LinkedIn Spring Extensions (Components)

```
…
<lin:component
    id="remoteContentCommunicationService"
    location="comm-server-client-cmpt">
    <lin:wire property-name="activemq.producer.brokerURL"
                                property-
value="${activemq.producer.brokerURL}"/>
    <lin:wire property-name="comm.server.httpRpc.url"
                                property-
value="${leo.comm.server.httpRpc.url}"/>
</lin:component>
…
```