"Linked Open Apps Ecosystem to open up innovation in smart cities"
Project Number: 297363

Deliverable:	**Evaluation Metrics**
Version:	**1.5**
Delivery date:	**08/10/2013**
Dissemination level:	**PU**
Author:	**Johan Verelst (CISCO)**
	Frank Van Steenwinkel (CISCO)

Summary:

This report presented a set of metrics and indicators to monitor the quality and effectiveness of the proposed architecture in real deployment scenarios

This report is a guideline to develop the iCity TEST Plan.

Metrics included are essential, ***but not limited to the ones presented.***

TABLE OF CONTENTS

1. Introduction

This report presents a set of metrics and indicators to monitor the quality and effectiveness of the proposed architecture for the iCity Platform in real deployment scenarios.

The following areas are addressed:

- Security
    - XML-based Web services are becoming a more pervasive foundation technology for integrating applications and exchanging data in Service Oriented Architectures (SOAs). Like all new technologies, however, XML-based Web services also present new security challenges in the form of XML data structures, granular application calls, input data, or executable attachments, all of which can be maliciously constructed to damage or expose a receiving application. XML-based Web services compound the number of vulnerabilities by providing access to application APIs and target applications. The distributed, peer-to-peer nature of Web services also introduces bilateral threats and vulnerabilities that can be passed through multiple application hops. A complete threat-protection framework needs to address three key functions: Prevention, Protection, and Screening.
- Operations and management capabilities
    - Different user roles need to be identified on the iCity Platform. Several Administrative account levels need to be able to deal with areas as user management, branding the portal, publishing API's. This will allow developers to smoothly build their applications.
- Reporting
    - Developers should have the ability to understand what their user experience is like, to track API performance and to ensure SLAs are being adhered to.
    - Administrators should be able to track the performance and usage of their APIs and applications built against these APIs, directly from within the Portal.
- Performance testing metrics
    - Key metrics need to be identified as the most important statistics that are reported on.

2. Security Aspects

2.1 API Management and Security

Publishing APIs online makes organizations subject to the growing threat of cyber-attacks. While network firewalls can provide some measure of protection from standard, Web-based attacks, they cannot address API threats because they lack the ability to deal with the messaging protocols used by APIs today, such as XML and JSON.

Managing APIs also presents a number of problems, primarily around creating, maintaining and updating different versions of APIs for different customers, as well as granting third parties the ability to aggregate and orchestrate across APIs to create new services and richer responses to queries. APIs are like any other piece of code that is created; they are developed, tested, deployed and revised as needed. However, moving Web APIs between environments or deploying new versions of APIs can expose hidden dependency issues or break your customers' existing integrations, causing downtime or even SLA violations. The iCity Platform should protect against attack and downtime.

2.2 Communication between the iCity Gateway and the back end server

The communication between the Gateway and the back end server needs to be secured.

Authentication and Authorization of API access should be done only by the Gateway (with developer management provided through the API portal). Protecting the API is the job of the Gateway, not of the back end server.

To avoid many back end system security reconfigurations, and to centralize the security rules, the Gateway (not individual developers) authenticates itself against the back end system.

Fixed 'username/password' basic authentication over HTTPS is probably the most simple method and might be acceptable at the initial stage. In other words we would assign the Gateway a username/password with which it authenticates itself against the back end server. Since basic authentication itself is not encrypted (only obfuscated through base64 encoding), it is strongly advised to use it over HTTPS. In that way at least the password is protected during transport.

2.3 Authenticating an API

The iCity Platform must have a utility to enable developers to discover APIs interactively.

By making choices between APIs' valid resources and methods, and then submitting queries and viewing responses, developers can, amongst other things, gain a better understanding of how the APIs work.

In order for a request to be executed correctly, an API must be authenticated on the iCity Gateway.

Different authentication methods should be made available, for example:

- API Key;
- HTTP Basic;
- OAuth 1.0;
- OAuth 2.0.

3. Operational and Management Capabilities

3.1 API Portal

The iCity API Portal enables cities to upload APIs, manage, and report on third party developers and the applications they build using the APIs.

The iCity Platform should contain these two main functional areas:

- API Portal: This area is used to publish APIs, manage developers, and perform analytics/reporting.
- CMS (Content Management System): This area is used to set up accounts, work with CSS files, configure system messages, and create Web content.

Administrator Account:

The iCity API Portal should have a predefined Administrator account that can be used to log in and create additional profiles via the content management system.

Changing a Password:

- Both developers and administrators should be able to change passwords.
    - Users can change their own passwords on the dashboard of the iCity API Portal.
    - Administrators can also use the CMS to change users' passwords, including their own.
- A maximum login attempt value should be provided.
- The time accounts are locked should be configurable in minutes, hours, days.

3.2 The Dashboard

- The iCity Platform should have a Dashboard; a primary interface for developers and several user roles pre-configured in the system: example: API Owners, Business Managers, and Account Managers.
- A navigation sidebar displays different links depending on the role of the logged in user; however, the Dashboard can be personalized by each individual user.
- Once you log in, the Dashboard page should be displayed by default.
- The iCity API Portal should provide the ability to create forums. Forums facilitate communication among all users in the API Portal. Forums should have different levels of access. Each level has permission to view different areas of the forum as well as perform different tasks.

3.3 Functionality by User Role

- The iCity API Portal should be delivered with several user roles pre-configured in the system.
- These roles should be defined as being either internal or external.
- Internal roles are created on the CMS and internal to business of implementing the portal, whereas external roles are accounts that must be invited to the system.
- Each user role should see different levels of the navigation to match their functionality.

3.3.1 Administrative Roles (Internal Roles):

The following “internal” user roles should be preconfigured in the iCity API Portal:

- **Administrator:**
  - The super user with access to all functionality for all the roles listed below:
- **WebAdmin:**
  - The person responsible for setting up the API Portal, including:
    - Branding the API Portal
    - Creating and publishing the home page, documentation, and other content
- **API Owner:**
  - The person tasked with defining, publishing and monetizing, or promoting the APIs. On the iCity API Portal, this person will be responsible for:
    - Defining API Plans (i.e., service levels) associated with each API
    - Publishing the APIs for use by developers
    - Measuring the effectiveness and usage of their APIs using the Analytics and Reporting feature

  The API Owner can also:
  - Manage organizations
  - Edit, enable, or disable applications
  - Email Organization Administrators
- **Business Manager:**
  - The person tasked with managing the developers who sign up to use the APIs. On the iCity API Portal, the Business Manager will be responsible for the following tasks:
    - Defining Account Plans (i.e., technical support levels) that can be assigned to each developer
    - Assigning Account Managers to developers
    - Measuring the rate at which developers sign up
    - Ensuring that SLAs (Service Level Agreements) are being adhered to, by using the Analytics and Reporting feature

  The Business Manager can also:
  - Process requests (such as application requests, API Plans, and Account registrations)
  - Manage organizations (the same as API Owners)
  - Edit email templates and registration disclaimers
- **Account Manager:**
  - The person tasked with assisting the Business Manager with the developers. On the iCity API Portal, this person will be responsible for the following tasks:
    - Approving API and Account plan requests

- Managing the developer's account on a daily basis
- Managing organizations (similar to an API Owner)

3.3.2 Developer Roles (External Roles)

The following "external" user roles should be preconfigured in the iCity API Portal:

- **OrgAdmin:**
  - OrgAdmin is the owner of an organization. This is typically a third-party user that signs up for an account in the iCity API Portal using the Registration Form. This person is responsible for managing his or her own organization and is usually the only developer or the first one to register for the organization.
- **Developer:**
  - A user that has been invited to join the iCity API Portal by an organization owner (OrgAdmin). These users are enrolled under the OrgAdmin's account. Developers are responsible for creating and managing new applications.

3.4 Tasks Performed by User Role

The following table summarizes the tasks each user should be able to perform

	API Owner	**Business Manager**	**Account Manager**	**Developer**	**Web Admin**	**Administrators**
View APIs	X					X
Publish APIs	X					X
Use or Designate Private APIs	X	X	X			X
Deprecate APIs	X	X	X			X
Add/Edit API EULAs	X					X
View and Message OrgAdmin	X					X
Create and Manage Account Plans		X				X
Request Account Plan Change				X		
Manage Account Managers		X				X
Manage Organizations	X (for	X	X (only	X (if		X

(Access varies by user role)	APIs)	(all orgs)	assigned orgs)	OrgAdmin)		(all orgs)
Manage or Work with Applications (Access varies by user role)	X	X	X	X		X

4. Reporting

4.1 Accessing Reports

- Administrators should be able to generate reports on APIs and organizations.
- Administrators should be able to access ranked reports.
- Developers should be able to generate reports on their applications.
- Developers with access to reports should be able to view usage reports.

4.2 Developer Reports

Developers should have access to both Application reports and API reports.

Usage and Latency reporting is recommended.

The Application reports should allow developers to:

- **View the usage graph for an application**
- **View the latency for an application**

The API reports should allow developers to:

- **View the usage for an API**
- **View the latency for an API**

4.3 Publisher Reports (Administrators)

Publishers should have access to both Application reports and API reports.

Usage and Latency is not only important for developers, but for publishers too as this will indicate somehow how end users will experience the application.

The Application reports should allow publishers to:

- **View the usage for an application**
- **View the latency for an application**

The API reports should allow publishers to:

- **View the usage for an API**
- **View the latency for an API**

4.4 Usage Reports (Administrators)

The usage report should provide a high level view of the Account Plan usage by organization.

4.5 Ranked Reports (Administrators)

A Ranked Reports page should be available to all internal user roles that have access to Administrators, Business Managers, API Owners, and Account Managers.

This page provides a high level view of API usage or latency by application or organization.

- **Top Applications:**
  - View the applications that have the most hits against APIs. These are going to be the most popular applications.
- **Highest Latency:**
  - View the applications with the highest latency spikes over the time period

chosen. The information here can help to perform troubleshooting.

- **Top Organizations:**
  - View the organizations that have the most hits against APIs. These are going to be the most valuable organizations.
- **Inactive Organizations:**
  - View those organizations that have generated no traffic or are no longer active on the Portal. These are accounts which might be purged.

	API Owner	**Business Manager**	**Account Manager**	**Developer**	**Web Admin**	**Administrators**
Manage or Work with Applications (Access depends on user type)	X	X	X	X		X
Approve/reject New Accounts		X				X
Approve/reject API Plan Requests		X	X			X
Assign Private API Access (to Developers)		X				X
Register for an account				X		
Add new applications		X	X	X		X
Access the Site Settings		X (email Templates and Registration only)			X	X
Access the Content Management System (CMS)	X				X	X

5. Performance Testing

Benchmarking web services usually involved simulating lots of users and sending lots of messages to simulate a heavy production situation.

Equally important is the desire to understand what metrics are available and the relationship between them.

5.1 Key Metrics

5.1.1 Requests per Second

In terms of overall statistics, reporting systems often cover the number of requests served in a given month, or the worst case burst period request traffic. This implies metrics based on requests per unit of time and usually leads to throughput being regarded in terms of the number of requests per second.

Often requests per second are limited by networking issues:

- Network latency for very small messages can be a significant part of the whole request time overhead. This limits the number of new requests that can be accepted by the application stack in a given time period.
- Network bandwidth for large messages can limit requests per second as no more traffic can be put on the network interface. Modern hardware can easily swamp a Gigabit network with large messages.
- Some operations are time consuming and necessarily synchronous, like certain kinds of Lightweight Directory Access Protocol (LDAP) lookups, database queries, etc. Often, the only way to optimize this is to redesign the workflow to cache, or reduce these time wait cycles.

5.1.2 Bytes per Second

Some proposed benchmarks are based on measuring throughput in terms of bytes per second. For a variety of reasons these prove to be difficult to plan well. To accurately model a proposed application it is necessary to have knowledge of:

- Average incoming Message size
- Average back end Response time
- Maximum concurrency of back end systems
- Bottlenecks at Authorization and Authentication systems

5.1.3 Latency

Measuring the total elapsed time it takes a request to be serviced is critical in certain types of applications.

This has its own list of criteria that need to be taken into account:

- Usability of user interfaces is often enhanced with a faster response times.
- Latency becomes a performance benchmark especially in chatty applications that use a large number of requests to service a single user action.
- Technical people are often tasked with measuring this and the iCity Platform dashboard User Interface (UI) should feature instrumentation to show the separate components of request latency.

- It is important to note that latency and concurrency are often in opposition when building test cases.
- External decision points like LDAP, Single Sign On (SSO) systems often contribute latency to a whole application.

5.1.4 Maximum Concurrency

Concurrency is defined as the number of requests being simultaneously processed at a given time. There are several states a request can be in:

- The initial TCP connection phases,
- Request message processing in the Gateway,
- Request servicing by a back end system, and
- The sending of the response back to the originating system.

Once a message reply is sent back to the requesting system, the Gateway resources are freed to process other requests.

Concurrency is usually the most misunderstood statistic in any performance discussion. This is covered in detail in the next section.

5.1.5 Number of users supported

This is rarely encountered. Mostly it is used interchangeably with maximum concurrency, though they do mean quite different things.

Often it means the concept of application and application firewall are somewhat intertwined.

In the next section a way to interrelate users, concurrency and latency will be described.

5.2 Concurrency, Number of Users and Latency

5.2.1 Planning

Planning for a good user experience and sizing the iCity Platform solution is a complex undertaking as there are different parameters as inputs and many ways of looking at the problem. Doing the calculation here is important to understand the issues.

One of the most common assumptions in sizing is that large concurrency is required to support a large number of simultaneous users interacting with the application. The usual mandate is to support your user base, and to plan to accommodate a worst case situation, so let’s see what real concurrency is needed by a large number of users.

5.2.2 The User Base

The following analysis is based on 20,000 users accessing a single application concurrently.

It is assumed that the application is web based, but has a core component that is sourced from some services component, i.e. the portal model. Most of the HTTP requests for a given page are things like images, CSS and other small static files and are serviced by web servers, not application servers, and so are not considered in this analysis. The calculations presented here are also applicable for fat client GUI-style applications because the same kind of technology choices around minimizing server round trips for heavyweight services also hold true for GUI applications.

5.2.3 Application Design

Designing an application to perform live queries for small pieces of user interface content is not good practice whether it is a client/server, a web app, or fat client. Waiting for even local network latency to fill in the content of UI elements like drop-down lists gives extra waiting

states during the displaying of the content. This make the UI appear unresponsive, and makes a large client base roll-out impractical for even unsecured applications, just from the sheer volume of the requests.

We are making a best practice assumption that services applications are designed to make one or two larger critical path requests as the core of the application service. We assume that most pages have a single type of information the user wants to view, but some will be more complex. We assume an average of 1.25 service requests per page view, reflecting a mix of page types.

5.2.4 Service Latency

Static content requests that require no processing should be sub millisecond in latency, but actual service requests are normally in the 10 milliseconds to 5000 milliseconds range on the back end. Later we describe how the service request latency is a hugely important number in determining required concurrency.

So far we therefore have 20,000 users, with 1.25 service requests per page, and each of those request taking from 10 to 5000 milliseconds to process.

5.2.5 Requests Per Second

Next we need to determine how many requests those users will generate.

Given the way that people read and use applications, the bare minimum time it takes to recognize a fully rendered page or UI, find the content he/she is looking for, then choose a navigation element to initiate another request is likely to be 3 to 5 seconds. That is the bare minimum. The time that users are **not** generating new requests to back end services is called the **page dwell time**.

Dwell time on a page of something like traffic information, a purchase order or a line of business task like a shipping request is going to be longer than 5 seconds.

So, given a page dwell time between 5 and 60 seconds, over the course of an hour, 20,000 users are going to generate between 0.75 and 18 million requests, or between 208 and 5,000 requests per second. This is a reasonable number for the requests per second statistic, but leads us into the discussion of needed concurrency and how latency is by far the critical statistic.

Requests Per Page	Page Dwell Time Min (seconds)	Page Dwell Max (seconds)	Request Service Latency (milliseconds)	Requests / Second Max	Requests / Second Min	Required Concurrency Max	Required Concurrency Min
1.25	5.0	120	10	5,000	208	50	2
1.25	5.0	120	20	5,000	208	100	4
1.25	5.0	120	100	5,000	208	500	21
1.25	5.0	120	1,000	5,000	208	5,000	208
1.25	5.0	120	2,000	5,000	208	10,000	417
1.25	7.5	120	5,000	3,333	208	16,667	1,042

5.2.6 Concurrency Calculations

The calculation for the required concurrency is as follows: 20,000 users generating 1.25 service requests per page every 5 seconds would generate, on average 20K * 1.25 * (5/60) or 30,000 requests per minute or 5,000 requests per second. We need to handle 5,000 requests every second and the service takes 10 milliseconds to handle a single request. In one second there are 100 periods of 10 milliseconds, so in each of these 10 millisecond periods we need to retire 5,000/100 or 50 simultaneous requests.

Required Concurrency=Requests per second / (1/Latency in seconds)

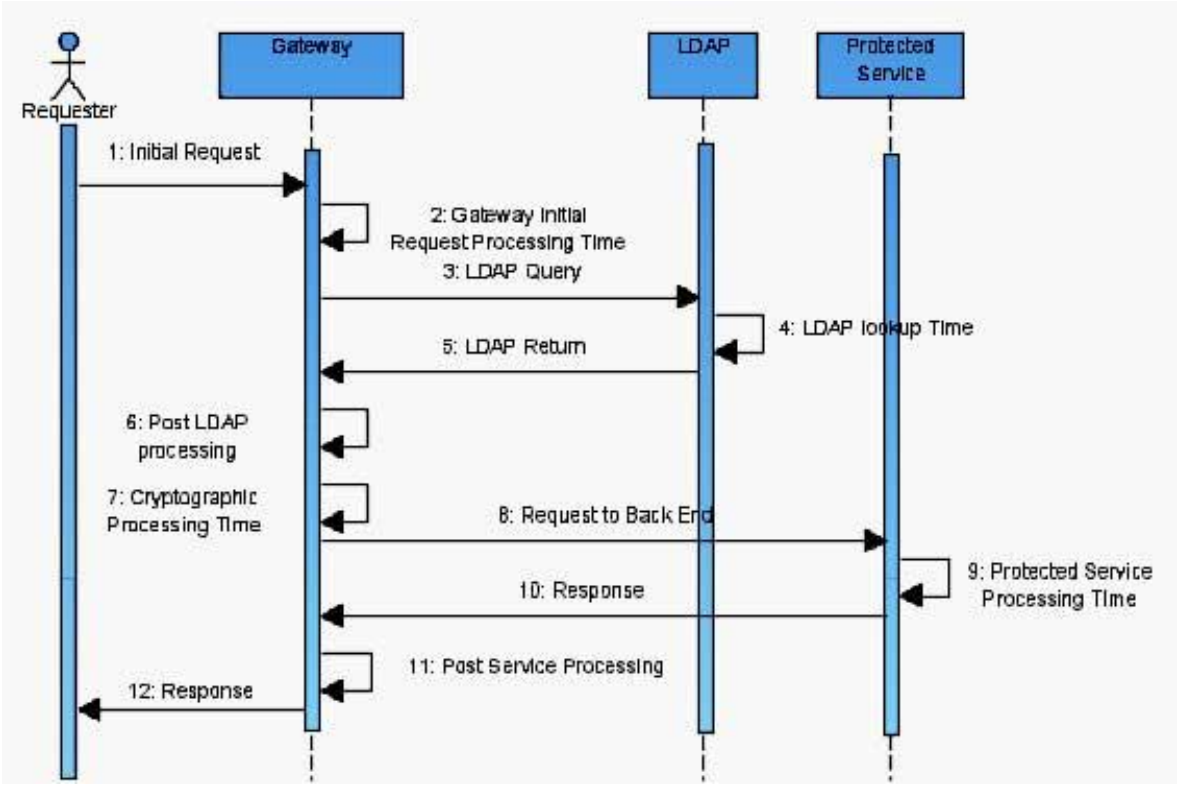
In our first example 5,000/(1/0.010) = 5,000/100 = 50. Of very important emphasis here is the effect of latency on concurrency. Assuming a lower page dwell time of only 5 seconds, and starting from 10 to 5,000 milliseconds service latency, the concurrency requirement jumps from only 50 simultaneous requests per second required to service 20,000 users to 16,667. At this point the performance of the system will seriously deteriorate, because at 1.25 requests per page, it would take an average of 7.5 seconds just to make the data available to render the page.

There are a large number of simplifications in this calculation but it does demonstrate that characterizing the load and the user experience has a huge impact on a prediction of required concurrency. How long will users wait for data before they decide the system is too slow?

Requests per user action also has a direct relationship to concurrency. Less clear is the effect of page dwell time. These worst case numbers reflect a given user, on average, asking for new content every 5 seconds. That may be considered as fast for most pages, unless the system has been built with lots of paging through content. Then, if what they need is on page 3, they may not wait 5 seconds to ask for new content. This can create a worst case scenario unintentionally as user acceptance testing may not accurately reflect how often people generate new requests, because the environments often are not loaded with enough data to require paging through content.

Latency Is Key

Sequence Diagram



Latency is inversely proportionate to needed concurrency

In the discussion of concurrency we described an application analysis with total application service latency as a huge determining factor in concurrency requirements. There are many contributors to latency, and the Gateway function needs to be monitored in that respect.

The above sequence diagram describes the processing steps and messages, internal lookup requests and points of latency when servicing a single inbound request at the Gateway

function.

Some systems specifically report the time between steps 1 and 12 as the front end response time and the time between 8 and 10 as the back end response time.

Experience has shown us that those are the two most important items to report when measuring latency.

Of note in this example is that the maximum front end response time or more accurately, the latency experienced by the end user was only 132 milliseconds even though the back end response time was 100 milliseconds.

5.2.7 Back End Processing

In almost all scenarios we have encountered in the field, **the back end processing time produces the bulk of the latency**. This is beyond our control, but the iCity Platform can help by providing: an efficient requester subsystem, controls on concurrency and connection caching for SSL.

There are some components of overall latency that we end up classifying as "our local processing overhead". One of them, **LDAP Lookup Time** is minimized somewhat by our authentication cache, but still can be a limiting factor. This call to LDAP has similar analogies in Single Sign On authorizations and other methods of external decision point references. This latency is not separately described in our UI, and may in some cases result in the Gateway itself suspected as being a source of latency.

Also of particular impact is cryptography. Cryptographic operations can incur latency and/or heavy CPU usage depending on the use of internal Hierarchical Storage Management (HSM), internal software cryptography or external HSM solutions. We have very efficient cryptographic capabilities, but there is an associated mathematical complexity associated with public key operations that no system can avoid.

5.2.8 Performance Optimization

With back end latency so dominating normal performance testing, the iCity Platform should be optimized to minimize delay in back end processing.

Small messages have given typical processing rates of 20,000 requests per second, for latency in the sub-millisecond range, so in most cases, the Gateway is not contributing any significant amount to latency.

Some policy elements have latency associated and can be avoided in latency sensitive applications; Auditing is the obvious one as it has dependencies associated with synchronously waiting for the auditing subsystem to write to hard disk.