Implementation and Evaluation of a Component-Based framework for Internet Applications

Mark Wallis, Frans Henskens, Michael Hannaford
School of Electrical Engineering and Computer Science
University of Newcastle
Newcastle, Australia
mark.wallis@newcastle.edu.au

David Paul
School of Science and Technology
University of New England
Armidale, Australia

***Abstract*—In previous publications we have introduced the concept of using a component-based software engineering paradigm to build Internet-enabled applications. We have proposed that this design allows for greater flexibility in deployment, better utilisation of resources and a reduction in total application development effort. We have described a system and realised that system as an API that can be used to design, build and execute such components. In this report we provide an overview of the key system components and present an implementation of an application developed using the system. We use this application to perform experimental and functional comparisons to show that the system provides advancements over the status quo.**

***Keywords*—cloud computing; component-based software engineering; platform-as-a-service**

I. Introduction

Component-based software engineering [1] is a software design paradigm focusing on loosely coupled, functionally distinct executing components that are orchestrated together to form a working system. Traditionally, these components all executed on a single host. The move into distributed component-based software engineering saw a paradigm shift that allow components executing on multiple hosts to function together, over a network, to form a working application. The network connecting the components was generally an Intranet, or a limited Internet, joining a small number of finite components. Cloud Computing [2] saw further evolution of distributed computing by providing a remote platform for the execution of software, including software components. When a component execution environment is deployed on Cloud resources it is generally viewed as a platform-as-a- service (PAAS) model.

If a software engineer wishes to build an application with multiple components executing in multiple cloud environments, with the current distributed component-based and Cloud Computing platforms, then the approaches are cumbersome. The developer is required to identify the end points of the inter-platform connectivity and write code specific to each Cloud environment that performs the integration task. For example, the developer may choose to implement a web service [3] “server” component in one Cloud, and a web service “client” component in another Cloud in order to build a connection between the two components. This approach does not scale well when you have a many-to- many relationship between components in multiple distinct Clouds that wish to communicate with each other.

Our previous research [4, 5] presented an approach that addresses this issue by introducing a scalable service-bus style architecture that offloads the issues of locating, communicating between and securing communication between components to a set of dedicated system-level components. This allows components to communicate with each other across non-homogenous Cloud environments without the developer having to be specifically aware of the deployment scenario.

This paper presents an example implementation of an application developed using this component-based system. Functional and experimental comparisons are made to show how developing Internet-enabled applications using this solution provides benefits over traditional approaches.

II. System Overview

Our previous research [4] has addressed the issue of how to deploy component-based software solutions across a varied set of resources which form part of a larger execution environment. The execution environment can include a mixture of personal devices (such as workstations and smart phones), servers (including private data centres) and public Cloud Computing. The Cloud environment support relies on infrastructure-as-a-service resources (IAAS) being deployed to execute our tailored runtime environment. Fig. 1 depicts a standard deployment of multiple components across multiple compute resources which work together to form a working application.

A strong focus has been placed on ensuring that the software developer is abstracted away from implementation specifics, such as having to know where specific components are executing and what resources are available for that user. A name-space approach has been taken to segregate available

Copyright
© Wallis, Henskens, Hannaford, and Paul 2017

ISSN (Print): 2204-0595
ISSN (Online): 2203-1731

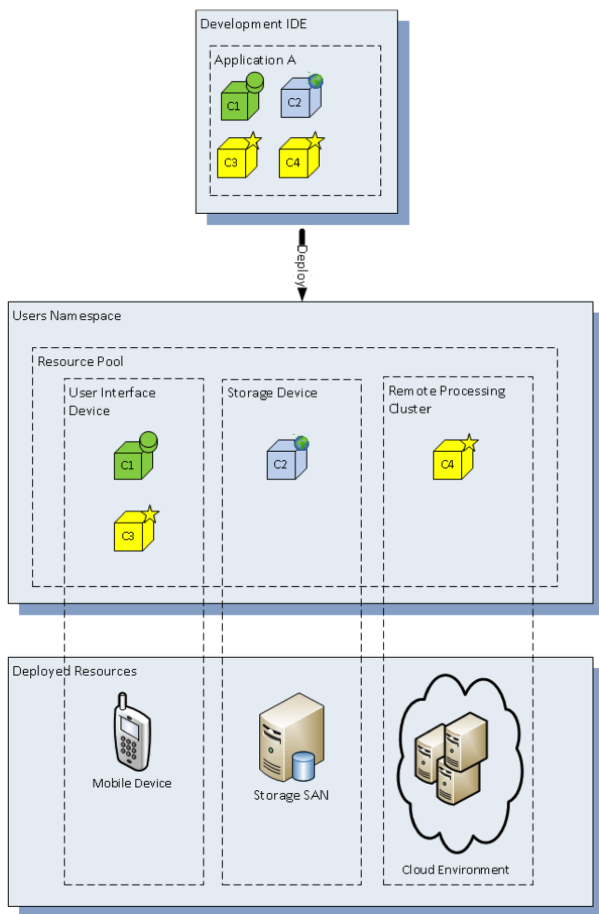

Fig. 1. Deployment overview.

resources in a way that ensures users can access any compute and storage resources available to them, while also providing protection from malicious access.

A. *Application Bootstrapping*

Each application in this component-based environment is defined by an "application XML" file. This file lists the components required to execute that specific application. Application XML files can either be stored and loaded from a local cache on a user's PC, or accessed via a uniform resource identifier (URI). MIME types and web browser plugins provide a direct means for users to execute applications simply by clicking a link on a website or selecting a bookmark, in much the same way that HTML5 [6] offline applications execute. The bootstrapping triggered by a user's request to start an application loads the application XML and completes the necessary pre-work required to allow the execution of that application. These pre-work steps are fulfilled by system-level components providing services such as component resolution.

A key focus of our research is ensuring that what is provided is an open API, rather than another distinct platform as a service environment. Our research describes the overall system architecture, including the set of system-level components required to facilitate application execution. It is left to the specific implementation of this API to make decisions around how the system-level components will internally function. This ensures that the API is open and capable of being implemented as a wrapper around existing platform-as-a-service models. A prototype implementation of the system-level components has been built as part of this research to provide a basis for comparison with existing distributed web application development approaches.

B. *System Level Components*

The API defined by our research permits implementation of a lazy-loading approach to component resolution. Application components are defined as having an initial state, with the options being:

- Mandatory - the component must be available in the user's namespace during application bootstrapping.
- Lazy - the component can be resolved at a later date when the first call to it is made from another component within the application.

No matter if the component's state is mandatory or lazy, the resolution of the component is handled by the system-level "component locality service" component. This component is currently responsible for extracting the global unique ID (GUID) for that component from the application XML and resolving it to a URI which the "component loader service" component can then use to download and instantiate the component within the environment. As discussed previously, the process followed to complete this resolution step is left for the implementation, however the API implementor sees fit. Implementations during prototyping of the solution have used a global component directory. Once the URI is obtained the "component loader service" component must connect to the URI to download the specific version of the component requested. At this stage, two actions can occur:

- The compute resource bootstrapping the application is online and able to access the URI. The component is downloaded and instantiated.
- The compute resource bootstrapping the application is offline, or online in a partitioned state such that it is unable to access the URI. The component is then marked as unavailable. If the component initial state is marked as "mandatory" in the application definition then the application fails to bootstrap. If the component initial state is marked as lazy then the component that made the initial call that triggered the component instantiation will receive an exception.

To facilitate these actions there are a number of system level components that are expected to be provided by the runtime environment. These include the following:

Application Bootstrapper - responsible for loading the application XML and identifying the set of mandatory components required to execute the application.

- Component Locality Service - responsible for receiving requests for references to components, locating the component in the relevant namespace and returning a valid reference. The reference will either be pointing to

Copyright
© Wallis, Henskens, Hannaford, and Paul 2017

ISSN (Print): 2204-0595
ISSN (Online): 2203-1731

an existing instantiated copy of the component within the users namespace, or a download URI which the component loader service can use to download the component code module and instantiate a new instance of the requested component.

- Component Loader Service - responsible for taking an un-instantiated component reference, downloading the component code module and instantiating it into a local stub object usable by the calling component
- Component Cache Service - responsible for storing cached versions of component code modules in order to speed up bootstrapping and support offline execution.
- Component Execution Service - a programming language-specific service that can take a component code module and physically execute it on a host. This is the key building block of allowing an application to be built out of components written in different programming languages.
- Component Communication Service - responsible for implementing the inter-component service bus on a local host and passing messages to other instances of the communication service on other hosts as required.

A key feature of the API presented in our research is that software engineers can implement their own versions of the system-level components, such as the “component loader service” component. This gives the overall solution the ability to be retrofitted and wrapped around existing component-based solutions and reduces the risk that the solution is to become yet another competing Platform-as-a-service (PAAS) environment [2].

The issue of updating components within the component cache service is addressed by using a version tag in the application definition. If a developer wishes to deploy a newer version of a component they can issue an updated application XML definition file to the user. The “component cache service” component uses the component name and the component version as a key into the cache store. If a newer version of a component is requested then the cache lookup will fail and the “component loader service” component can then obtain the newer component via the appropriate means.

A diagram showing this bootstrapping workflow is given in Fig. 2.

C. Application Components

For a developer to build a solution using this framework the following development process is followed:

1. An application XML file is created defining the application name and version.
2. The initial application component is built. This component is responsible for starting the application and often provides the user interface, and its inclusion is mandatory to the application XML.
3. Any other components which are required by the initial application component are then built. Depending on the configuration these components can be added to the application XML as either mandatory or lazy-loading.
4. For each new component being created the interface must be defined. The interface can either be chosen from a library of pre-existing interfaces (supporting reusability and loose-coupling) or created manually for this specific component. The former is promoted as the preferred option.
5. All components are then registered and made available. As discussed above, during the prototyping phase of this research availability was achieved by registering the component in the “global component directory”.
6. The application XML is then distributed to the end user for execution.

For an end user to be able to execute the application there are a number of prerequisites:

- The initial host bootstrapping the application must have access to the application XML file.
- The runtime environment must be installed on the device being used to bootstrap the environment, and any device that wishes to host the execution of components.
- The runtime environment must contain a “component execution service” component for each programming language required by a component of the application, as discussed below. These should be listed as mandatory components within the application XML and are resolved just like any other non-system component in the environment.

D. Building, Finding and Instantiating Components

The “component loader service” component has a number of key responsibilities. If the “component locality service” component passes a reference back to an existing instantiated component then it just needs to translate that reference into a stub object which the calling component can then use. If a download URI reference is provided instead then the “component loader service” component must download the component code module and work with the “component cache service” component in order to instantiate a new copy of the component. After the component is instantiated it is then registered in the user’s namespace in the “global component directory” so future requests can be resolved to this executing copy.

A component reference is made up of two parts - a URI pointing to the instance of the “component communication service” component which is executing on the same host as the requested component and a GUID that represents that specific component instance. Whether the “component locality service” returns a reference to an existing executing component or the download URI to instantiate a new instance is based on a number of factors;

- If no reference to the component can be found in the global component directory then an error is returned. This can occur if the developer fails to register the component correctly during the development cycle.

Copyright
© Wallis, Henskens, Hannaford, and Paul 2017

ISSN (Print): 2204-0595
ISSN (Online): 2203-1731

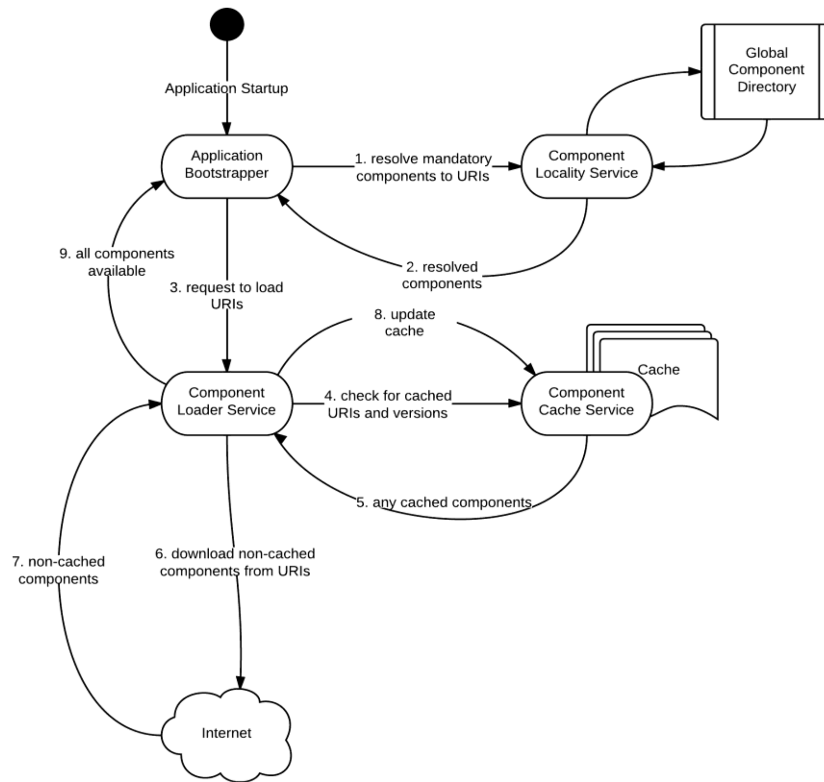

Fig. 2. Boostrapping workflow.

- If the component is located in the directory but no instantiated instances are registered then a download URI is returned. This download URI is provided by the developer during the registration process.
- If an instance of the component is located then the component's "scope" value is checked. The scope of a component is defined by the developer during the registration process and is discussed further below. If the component scope is valid for the inbound request then an instance reference is returned, otherwise the download URI is returned.

The application developer defines the scope of each component during the registration process. The following options are valid:

1. A "global" scope means that the environment framework will only ever instantiate a single copy of that component. The component will be owned by the developer and all users wishing to access that component will access a shared instance. This option allows the system to provide a "software-as-a-service" style component model.
2. A "user" scope means that the environment framework will instantiate a copy of the component per requesting user namespace. The component will be owned by the requestor and be shared across all requests coming from requestors within that same namespace.
3. A "request" scope means that the component will be instantiated for every request and then thrown away once de-referenced. The component will only exist for the duration of a single call.

E. *Inter-Component Communication*

The "component loader service" returns a stub back to the calling component in order to provide an abstraction between calling local components and remote components. If a call is made into the stub then that stub calls the "component communication service" component which is responsible for passing messages between components within the environment. Each physical host running a copy of the execution environment has a running copy of the "component communication service". This service acts as a local service bus for inter-component communication and an uplink to other running instances of the communication service that are referenced by the "component locality service" component.

It is the choice of the "component loader service" component where to physically instantiate components within the user's namespace. By default new components are instantiated on the same host as the requesting component. An option exists, though for the "component loader service" component to pass the instantiation request to another instance

Copyright
© Wallis, Henskens, Hannaford, and Paul 2017

ISSN (Print): 2204-0595
ISSN (Online): 2203-1731

of the "component loader service" component attached to another host within that user's namespace. This is achieved by passing the request to the "component communication service" component on that remote host. Resolution is then completed using the above process on the remote host and a reference is passed back between the communication services. Future access then is transparent and handled by the "component communication service" component.

F. Review

The system described above is intentionally light-weight. Decisions on where to instantiate new components and how to store a directory of instantiated components within a user's namespace, and the global namespace, are left to the implementations of the various system level components. The aim of this research is to provide an open API for implementing a global component-based application framework that allows developers to tailor the core environment to their specific needs. For example, one application could execute within a user's environment where the user is running a "component locality service" component that relies on a global directory and without any change, that same application could execute within a user's environment where the "component locality service" component relied on a massively distributed hash map for storing component references. The abstraction of system-level components provides flexibility, a path for growth, and the ability to retrofit this new framework on top of existing component-based software solutions and Cloud platform-as-a-service environments.

III. IMPLEMENTATIONS

To prove the viability of the system a prototype implementation has been developed. This involved building prototype versions of the six system-level components and then building an application on top of that environment. An application was chosen that already had a Web 2.0 implementation in order to provide a method for comparison and experimentation.

The application chosen to base the prototype on is QuON [7]. QuON is a typical Web 2.0 MVC [8] application which provides users with the ability to design, deploy and complete online surveys. The surveys are defined by researchers and the QuON system has been tailored to allow maximum flexibility and customisation. Custom branding, custom question types and custom output formats are all supported, including external reporting and system integrations. Deployment of QuON has followed a typical LAMP process of Apache web server, Linux operating system, MySQL database and PHP execution environment. QuON is run as a software-as-a-service model with the server-side hosted by the University of Newcastle. Clients interact with the server using a web browser. The web browser is fed Javascript and HTML5 compliant client-side code. QuON is used heavily by researchers in the health-behaviour research space which requires stringent ethics approval and regulation.

A number of limiting factors have been identified with the existing Web 2.0 QuON application:

- There is no capability for a survey to be taken while the end user is offline. The user must remain online and connected to the Internet throughout the duration of the survey in order for their web browser to be able to communicate with the server hosting the QuON web application.
- To execute the survey the user must have access to a web browser. This limits the client to completing the survey on devices with a standards-compliant, up-to-date, web browser. As an example, there is no support for having the user complete the survey using a voice-enabled client or a native mobile-app. The current QuON mobile solution relies on a mobile web browser which does not provide the most optimal user experience when compared to a native mobile-app.
- Persistence of survey results is limited to a single MySQL database. There is no capability of allowing researchers to store their survey result sets in their own data stores. This opens up data ownership and privacy concerns which are a concern during the ethics approval process.

The goal of the prototype was to address the above three concerns using the presented distributed component-based execution environment, while showing that distributed component execution could be achieved with little-to-no effort by the application developers. Focus was also given to the possibility of porting existing software-as-a-service applications over to the new component-based framework.

The first part of the prototype focused on the implementation of the required system-level components. The following system level components were developed:

- Application Bootstrapper - a Java boot loader capable of kickstarting applications.
- Component Locality Service - a Java locality service which interacted with a single global component directory.
- Component Loader Service - a Java loader service that was capable of handling both instantiated references and downloadable URI requests.
- Component Cache Service - a Java component cache service that was able to cache local copies of the component code modules.
- Java Component Execution Service - a Java component execution service which can execute components written in the Java programming language, including both system and application components.
- LAMP Component Execution Service - a wrapper execution service which is capable of deploying an Apache web server, PHP, MySQL bundle. This is used to wrap existing legacy components of the application.
- HTML5 Component Execution Service - a service capable of executing an HTML5 based user interface

Copyright
© Wallis, Henskens, Hannaford, and Paul 2017

ISSN (Print): 2204-0595
ISSN (Online): 2203-1731

component, including managing a thin web browser on the users device.

- Component Communication Service - a service-bus component implemented on top of Apache ServiceMix [9].

With those system-level components in place the following QuON-specific application components were defined:

- QuON UI - a HTML5 component responsible for taking a survey definition, presenting a self-contained HTML5 user interface and generating a survey result set message
- QuON Survey - a LAMP component which wrapped the existing "controllers" from the legacy QuON MVC implementation.
- QuON Data Store - a Java component responsible for persisting survey result sets and survey definitions.

The QuON UI component was evolved from the existing "view" layer, taken from the legacy MVC application. The HTML was re-worked to provide a self-contained HTML5 experience that integrated back to the greater application framework using a REST communication channel. The QuON Survey component contained the "controllers" from the legacy MVC application, extracted from the legacy application and wrapped in a component interface. The QuON Data Store component replaced the "model" layer from the legacy MVC application and acted as an object store for survey definitions and survey result sets.

These three components were orchestrated into two separate applications:

- The "QuON Server" application contained both the "QuON Survey" and "QuON Data Store" components. This allowed research groups to instantiate their own copies of the survey engine and data store.
- The "QuON Client" application which allowed end users to request surveys from a "QuON Server" application. Multiple "QuON Client" applications can connect to the same "QuON Survey" application.

The following high-level application flow was established:

1. The researcher bootstrapped the "QuON Server" application on their own server. This registered the instance of the "QuON Survey" component within the global namespace. As part of the component startup the researcher is provided with the GUID for the "QuON Survey" component.
2. The researcher distributes a link to the "QuON Client" application XML to the end users. Part of this URL is a parameter referencing the GUID of the "QuON Survey" component instantiated above. This parameter is passed through the application bootstrap process to the "QuON UI" component.
3. The end user bootstraps the "QuON Client" application on their local machine. The "QuON UI" component communicates with the "QuON Survey" component over the service bus provided via the "Component Communication Service" component and requests a list of the available surveys.
4. The surveys list is provided to the end user where they select the survey they wish to complete. This triggers another message from the "QuON UI" component to the "QuON Survey" component requesting the survey definition.
5. The "QuON UI" component provides the survey to the end user to complete. At the completion of the survey a survey result set is generated.
6. The "QuON UI" component passes the survey result set to the "QuON Survey" component which in turn passes it to the "QuON Data Store" component for persistent storage.

IV. EVALUATIONS

The primary aim of the prototype described above was to provide a platform for both experimental and functional comparisons between the new distributed component-based framework and existing web application approaches. The evaluation has been broken down into performance analysis, functional comparison and software development process review.

A. *Performance*

End-to-end performance was compared between the two solutions on the basis of round-trip latency. Latency was measured at application startup and also through a typical end-to-end user flow.

Using the legacy Web 2.0 application, startup was measured by timing from starting the web browser through to when the initial survey page was displayed to the end user. Application startup latency with the distributed component-based architecture was measured from the start of application bootstrap until the time where the initial survey page was displayed to the end user. Over a series of 100 executions the averages shown in Fig. 3 were collected. It can be seen that initial startup performance was worse under the new distributed component-based architecture upon initial application startup. This is due to all the components needing to be resolved and loaded. On further application startups, though, the average is only marginally worse than the legacy solution due to local component caching.

The next performance metric measured was the round-trip time taken to progress through 20 survey questions. This involved automating the answer submission process with a delay of 1 second per question. Over a series of 100 executions the averages shown in Fig. 4 were collected. As shown, the round-trip time was improved with the component-based solution. This can be explained by the fact that all communication between components when moving from one question to the next is all local to the "QuON UI" component, whereas with the Web 2.0 solution a HTTP request/response round trip to the server is required each time. There is an initial performance hit at the start of the process while the survey

Copyright
© Wallis, Henskens, Hannaford, and Paul 2017

ISSN (Print): 2204-0595
ISSN (Online): 2203-1731

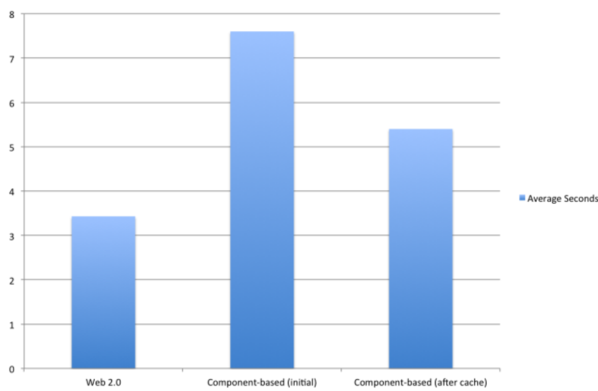

Fig. 3. Performance - startup.

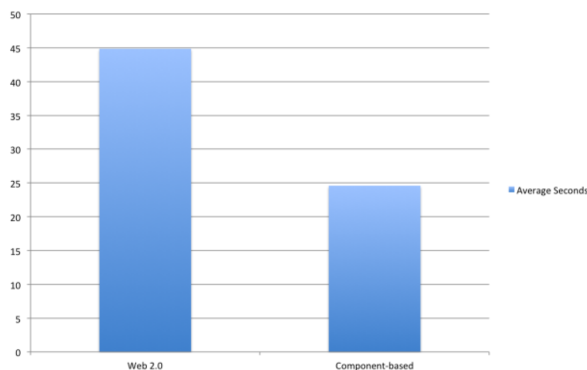

Fig. 4. Performance – 20 questions.

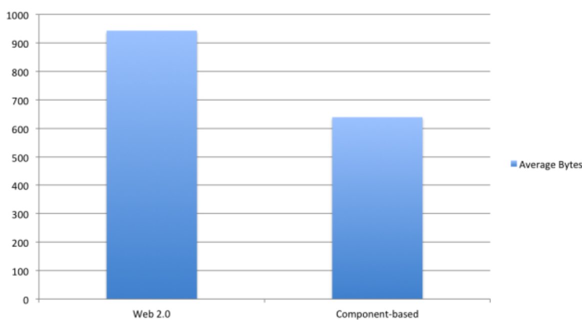

Fig. 5. Network data – 20 questions.

definition is loaded into the "QuON UI" component but after that the performance benefits are clear.

The network data exchanged between the various components of the solutions was also measured. The data shown in Fig. 5 was collected over a simulation used above with 100 executions over a 20 question survey. With the inclusion of component caching to the framework the bytes transferred was dramatically lower for the distributed component-based application when compared to the Web 2.0 application. This is especially true for subsequent survey runs after all survey definitions and components are cached. The reduction in meta-data being transferred with each survey question/answer pair (i.e. with data such as HTTP headers) also assisted in keeping the average bytes transferred lower for the Web 2.0 application.

B. *Functional*

A functional comparison between the Web 2.0 solution and the component-based solution was performed based on the three identified short comings of the Web 2.0 solution, documented above. Building the application in a distributed component-based manner allowed us to address the three main criticisms of the legacy QuON application listed above. Offline client support is provided assuming the client has bootstrapped the QuON Client application at least once in the past. The caching process ensures that a copy of the "QuON UI" component and all the system level components are cached locally. The "QuON UI" component also caches the survey definition. Any survey result set messages are passes to the "component communication service" component for asynchronous delivery once the client comes online again. While the "QuON UI" component in this prototype was implemented using a "HTML UI" executing component, the abstraction generated would easily allow a completely different user interface component to be developed. The loose coupling between the components allows the end user to easily switch between user interfaces. Packaging the "QuON server" as an application in its own right it allows the researchers to instantiate their own instances of the back-end, hence allowing for them to manage their own data storage and privacy.

From a user experience perspective the end user is required to have additional software installed on their PC in order for the distributed component-based application to execute. Parallels can be drawn though to the common requirement to have software such as a Java runtime environments (JRE) installed on client PCs, and hence is not seen to be a barrier to entry. Bootstrapping applications is performed easily by allowing the user to click on an application XML link in their web browser, and with the correct MIME type and application hooks configured within the browser, have it automatically bootstrap the application.

C. *Software Development*

From a software development perspective we have shown that it is possible to re-use existing Web 2.0 components within the new distributed component-based framework by wrapping them within component interfaces. We have also shown that, once the runtime environment and system-level components are available, the developer needs only be concerned with defining the interfaces and building the application-specific components required for their application. The runtime environment handles the complexity of the distributed nature in which the components are executing.

V. CONCLUSION

This research has shown that it is possible to build solutions using our previously published distributed component-based framework. A sample application is presented and used to provide functional and experimental comparisons between existing web technologies and the approach defined by this research. While the prototype may not be feature complete, it does show that once the base

Copyright
© Wallis, Henskens, Hannaford, and Paul 2017

ISSN (Print): 2204-0595
ISSN (Online): 2203-1731

environment is available the developer can build an application without needing to be concerned with the intricacies of inter-component distributed communication, on a massively distributed scale. A suitably advanced set of system-level components would allow for an end user to easily distribute their instantiated components over a set of hardware available to them within their namespace. The decision on where the components are instantiated could either be manual, or automated. Complexities presented by such manual choices are discussed in our previous work [10].

REFERENCES

[1] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley Professional, 1997.

[2] G. Boss, P. Malladi, D. Quan, L. Legregni, and H. Hall, (2007) Cloud computing. [Online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/wes/hipods/Cloud computing wp final 8Oct.pdf

[3] W3C. (2004) Web services architecture. [Online]. Available: http://www.w3.org/TR/ws-arch/

[4] M. Wallis, F. A. Henskens, and M. R. Hannaford, "Overcast skies - What cloud computing should be?" in 1st International Conference on Cloud Computing and Services Science (CLOSER 2011), 2011.

[5] ——, "The super-browser: A new paradigm for web applications." in INTERNET 2012: The Fourth International Conference on Evolving Internet (INTERNET), 2012.

[6] I. Hickson, HTML 5 Editor's Draft, w3c editor's draft 2011 ed., W3C, January 2011. [Online]. Available: http://dev.w3.org/html5/spec/Overview.html

[7] D. Paul, M. Wallis, F. Henskens, and K. Nolan, "Quon - A generic platform for the collation and sharing of web survey data," in 9th International Conference on Web Information Systems and Technologies (WEBIST), SCITEPRESS, May 2013.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.

[9] A. S. Foundation. (2015) Apache servicemix. [Online]. Available: http://servicemix.apache.org/

[10] M. Wallis, F. Henskens, and M. Hannaford, "Peer-based complex profile management In Software Engineering", Artificial Intelligence, Networking and Parallel/Distributed Computing, Springer-Verlag, 2011, pp. 103–111.

Copyright
© Wallis, Henskens, Hannaford, and Paul 2017

ISSN (Print): 2204-0595
ISSN (Online): 2203-1731