Maturity Model for Liquid Web Architectures

Andrea Gallidabino and Cesare Pautasso

Software Institute, Faculty of Informatics, USI Lugano, Switzerland
{name.surname}@usi.ch

Abstract. Liquid Web applications adapt to the set of connected devices and flow seamlessly between them following the user attention. As opposed to traditional centralised architectures, in which data and logic of the application resides entirely on a Web server, Liquid software needs decentralised or distributed architectures in order to achieve seamless application mobility between clients. By decomposing Web application architectures into layers, following the Model View Controller design pattern, we define a maturity model for Web application architectures evolving from classical *solid* applications deployed on single devices, to fully *liquid* applications deployed across multiple Web-enabled devices. The maturity model defines different levels based on where the application layers are deployed and how they migrate or synchronize their state across multiple devices. The goal of the maturity model described in this paper is to understand, control and describe how Web applications following the liquid user experience paradigm are designed and also provide Web developers with a gradual adoption path to evolve existing Web applications.

Keywords: Maturity Model, Liquid Web Architecture, Decentralised Web, Liquid Software

1 Introduction

The metaphor of liquid software [17,34] illustrates the user experience when interacting with software deployed across multiple devices. Liquid software can 1) adapt the user interface to the *set of* devices being concurrently used to run the application; 2) seamlessly migrate a running application across devices and 3) synchronize the state of the application distributed across two or more devices, effectively breaking down the continuity boundaries that exist between devices in proximity both in physical space as well as in cyberspace.

Web applications were traditionally developed following a thin client architecture whereby most of the logic and the entire persistent state of the application would be executed and stored on a central Web server. They would offer only partial support for the liquid user experience in terms of the ability of migrating the application by simply sharing the URL pointing to the current state of the application and adapting the user interface by employing responsive Web design techniques [24].

As the Web technology platform has evolved with enhanced support for rich and thick client architectures and for protocols beyond HTTP to enable real-time push notifications and peer to peer (browser to browser) connections, it is time to revisit the architectural design space of contemporary Web applications to systematically study new deployment configurations and how these impact the liquid user experience.

In this paper we present a maturity model for liquid Web applications [25], based on multiple facets that determine the degree of liquidity of a Web application both in terms of which liquid user experience primitives are enabled as well as how these can be implemented with different performance and privacy guarantees. Each architectural configuration presents unique challenges and opportunities to deliver a liquid behavior under different constraints. For example, while it is relatively easy to synchronize the state of the application relying on a highly available, centralised master copy deployed in the Cloud, some privacy and latency issues may warrant considering more decentralised or distributed approaches to data management.

The maturity model is based on three orthogonal facets, each having three levels: a) logic deployment (ultra-thin, thin, thick); b) state storage (centralised, decentralised and distributed); c) communication channel (HTTP, WebSockets and WebRTC). This paper provides a systematic discussion on the implications of the most significant architectural configurations on whether and how liquid Web applications can be built under the corresponding architectural constraints. Additionally, for each level, we survey existing Web development frameworks within the corresponding Web application architectures. As we are going to show, migration of Web applications can be achieved with all configurations, while cloning requires support for real-time synchronization that is only present in higher maturity levels.

2 Motivation

The relationship between users and their computing devices has evolved from multiple users sharing one expensive and large computer to the opposite, where multiple, cheap, mobile and Web-connected devices are owned by a single user. Web browsers are nowadays ubiquitous as they run on desktop computers, laptops, tablets, smart phones, digital cameras, smart televisions, cars, and – with some limitations – even kitchen refrigerators, watches and glasses. While responsive Web applications are designed to adapt to the different screen sizes and input/output capabilities of different devices, it is still challenging to develop rich Web applications which can seamlessly migrate across different user devices. For example, planning a trip on a large display and following the directions for the trip guided by the phone GPS; typing a short email by tapping on the phone screen but then as the email text grows longer deciding to continue the work on a computer with a real keyboard.

As users begin to use multiple devices concurrently – for example: to watch television while looking up information on their tablet, to play games across multiple telephones, to share pictures taken with personal devices and view them on a public display, to remotely control presentation slides from a watch, or to confirm a credit card transaction entered on a desktop computer using the fingerprint reader of the phone – only few Web applications fully take advantage of all available devices and distribute their user interface accordingly or allow users to re-arrange different user interface components at will.

Web developers and designers have successfully addressed the scalability challenge to serve Web applications to millions of users [1] and to personalize such applications to each user profile [6] (e.g., language, age, geographical location, regulatory constraints, etc.) and adapt it to the capabilities of their Web browsing device [24]. However, this has been under the assumption that users connect to use the Web application using one device at a time. Stateful Web applications which use cookies to establish a session with a particular Web browser make it difficult for users to switch devices in the middle of a browsing session [15]. Additionally, they may break when opening multiple tabs to run them and sometime assume that users logging in from different devices at the same time may indicate a security issue.

In this paper we provide a maturity model to assess how different Web application architectures can provide support for the liquid user experience for both sequential and parallel screening scenarios: – **Sequential screening**: users own more than one device, at any time they may decide to continue their work using another device. The application and the associated state seamlessly flow from one device to another; – **Parallel screening**: users own multiple devices and deploy the software on all of them. Users may decide to change the number of devices running the liquid application as well as to move components of the application from one device to another while keeping the state up to date.

Web developers can follow the maturity model to redesign, refactor and transform their applications to provide enhanced level of support for liquid behavior defining the following liquid UX primitives: – **Forward**: the ability of an application of redirecting the input/output of one device to another; – **Migrate**: the ability of moving a running application to another device; – **Fork**: the ability of creating a perfect copy of an existing application on another device. – **Clone**: the ability of creating a perfect copy of an existing application on another device, and keep the state of the original and the copy synchronised thereafter. Sequential screening can be achieved if an application defines either a migrate or fork primitive, while parallel screening can be supported either with clone or forwarding primitives.

3 Web architecture facets

Web applications comply with the client-server architectural style, in which persistent resources or services are provided by one server to multiple clients. Without loss of generality, we further describe Web application architectures using the Model-View-Controller (MVC) pattern, one of the most used design patterns in Web applications development [20]. In the MVC pattern Web applications are logically decomposed to manage separate concerns: data modeling and persistent storage, data processing and business logic, and data input/output and user interaction.

– The **Model Layer** manages the persistent data of an application. The model of a Web application also includes any of its assets such as Web Pages, images, and scripts that need to be transferred to the clients. This layer requires some kind of data storage able to represent, organize and collect information: • in the server-side of an application it usually takes the form of a database such as relational databases like *Oracle* and *MySQL*, document oriented databases like *MongoDB*, or *CouchDB*, or other schemaless databases like *Redis* [8]; • in the client-side usually the file system of the device is used as storage, but due to the possibility of having clients running on heterogeneous devices and implemented using different programming languages, data storages can highly differ from client to client even in the same application. *WebSQL*, *IndexedDB*, *LocalStorage*, and *Cookies* are standard implementations of data storage APIs available in HTML5-compliant Web browsers.

– The **Controller Layer** consists of the logic of an application. The controller layer is a bridge between the model and view layers, it manipulates data and executes tasks received from either layer and forwards the results to the appropriate one. Depending on where the controller layer is deployed it can be implemented using different programming languages. In the server-side *PHP*, *ASP.NET*, and *Javascript* (using *Node.js*) are the most used programming languages, while in the client-side *Javascript* is the main option.

– The **View Layer** is the graphical user interface of an application, consisting of the visual representation of the data and information retrieved from the model layer and rendered into an interactive visualization.

Combining the client/server execution environment and the three MVC layers, we identify different deployment combinations. While the View Layer is constrained to run on the client, both Model and Controller Layer can be deployed on either side (or partitioned to run on both client and server). Additionally, we distinguish three alternative communication channels and protocols (HTTP, WebSockets and WebRTC) used to interconnect the layers of Web applications running on different devices.

In the following sections we discuss more in detail each facet which will be combined into the liquid Web application maturity model in Section 4.

3.1 Model Layer deployment

Model layer deployment describes where the persistent state of the Web application is stored. We identify three levels based on whether data is logically centralised on the server or distributed towards the clients (Figure 1):

Level 1 - Centralised - The model is stored using any data management solution that is solely deployed in the server-side. For scalability and availability purposes, the actual storage can be implemented using multiple virtual servers running in a Cloud data center. Conceptually, this is still a centralised solution as data is never managed by the client. The advantage is that no matter what

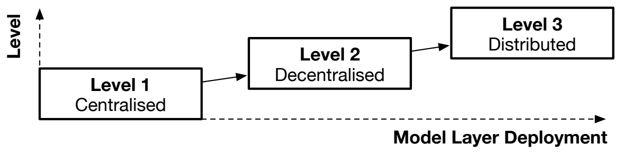

Fig. 1: Model layer deployment levels

client device is used to access it, the data will be readily available [32]. Users thus trade off the convenience of accessing "their" data anywhere with the loss of control over the actual location where it is stored and who else can access it. As clients always need to remotely request data from the server, there are also latency and availability implications to be considered. When multiple clients perform transactions to update shared resources, having a single master copy on the server helps to ensure consistency.
Some real world examples of centralised model layer deployments use databases created with *MySQL*, *MySQL Cluster*, or *Cassandra* [23].

Level 2 - Decentralised - The model layer is deployed both in the server and client-side of the Web application. Information stored in the server database is replicated or cached by the clients. Conversely, users may prefer to save the primary copy of their data in their own clients and use the server as a secondary backup.
Cookies are the simplest example of decentralised persistent storage on the Web. Web application using any technology mentioned in level 1 (e.g. *MySQL*) in combination with any HTML5 storage API (e.g. *localstorage, WebSQL*) falls in this category. *Apache CoudchDB* or *PouchDB* are databases that feature client-side caching with automatic synchronisation allowing offline availability of the retrieved data.
Decentralised approaches enhance: • data privacy, even though data must still be transmitted to the servers if there is not a direct communication channel between clients; • availability during offline operation, assuming the data has been prefetched by the client the Web application may still work while being disconnected from the server; • enhanced perceived performance when hitting data cached on the client.

Level 3 - Distributed - The model layer is distributed exclusively on the client-side of the Web application. There is no need to use the server to retrieve or store data, because clients completely own the state of the Web application. This positively impacts data privacy because the information of the users always remains on their devices and is never stored in a Web server outside of their control (e.g., in the Cloud).
Distributed model layer deployment can be achieved in a modern Web browser by using any combination of the storage APIs provided by the HTML5 standard, namely the *WebSQL*, *IndexedDB*, and *LocalStorage* APIs. On top of these technologies, or even using the File system of the devices running the client of the Web application it is possible to build distributed model layers able to automatically synchronise between clients (e.g. [36]).

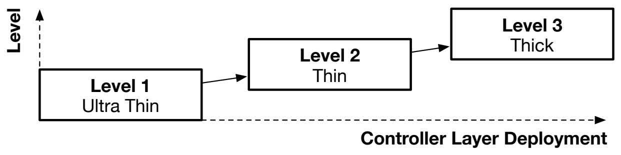

Fig. 2: Controller layer deployment levels

3.2 Controller Layer deployment

The Controller layer deployment determines where the Web application executes tasks and whether it can offload its workload. We define three levels with respect to the client thickness: (Figure 2):

Level 1 - Ultra-Thin Client - In this level the controller layer of an application is deployed only on the server-side of the application. The only logic present on the client is the logic needed to retrieve content from the server and to display views when they are received from the server.
Primitive Web browsers that did not allow running scripts, such as *JavaScript* or *Java Applets* can be seen as Ultra-Thin clients. Ultra thin clients always display the view layer statically and cannot adapt it to the client's device. *Curling* pages on a Terminal is also an example of Ultra-Thin Client, in which the forwarded raw data is displayed. Web applications that do not require scripts to run in the client fall in this category as well.

Level 2 - Thin Client - The logic of an application is deployed on both server and client-side of the Web application. The server can offload part of the computations to the connected clients. The most offloaded task in Web applications is the creation of the views which is entrusted directly to the clients needing it, however in thin clients any simple task can be offloaded to the clients. Whenever the client is thin, it is possible to make views responsive to the client's device. This allows the same applications to use a different look and feel in devices with different hardware specifications.
AngularJS, *React*, or *EmberJS* are some frameworks for isomorphic Web applications written in Javascript that require thin clients.

Level 3 - Thick Client - The logic of an application is entirely deployed on the client-side of the Web application. A big portion of the application computations are offloaded to the clients. As in level 2 clients compute the views they display. Additionally they execute computationally-heavy application-specific tasks that were not previously included in thin clients. The HTML5 WebWorker specification allows Web browser to run scripts in background, making it possible to develop complex client-side applications [7].
Thick clients can be aware of other connected clients, making it possible to adapt the view layer of the application on a set of devices instead of making it responsive to a single one. Complementary adaptive views can also automatically evolve in real-time if the application is able to propagate to all devices the knowledge of connections and disconnections of other clients.

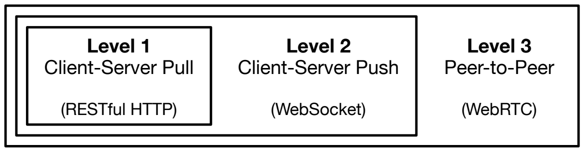

Fig. 3: Communication channels

Web 2.0 single-page applications generally require thick clients, any client described in level 2 can become a thick client if the entire controller logic layer is deployed in the client-side. Liquid Web applications featuring all the liquid user experience primitives require clients to be thick.

3.3 Communication channel

The communication channel facet is characterised by the direction of the communication between the client and the server and whether clients can communicate directly. The levels shown in Figure 3 are inclusive, whereby a higher level also includes all the features provided by the lower levels:

Level 1 - Client-Server Pull - Clients are always the origin of all request-response interactions with the server. Clients request resources addressed by URIs and the server responds with the corresponding representations if they exist. On the Web, this kind of communication is implemented with the HTTP protocol.

Applications relying solely on the HTTP protocol cannot propagate state changes or events occurring on the server back to the clients in real-time. They can only simulate a quasi-real-time environment (with continuous polling). While the liquid migrate primitive can be implemented with HTTP only, cloning or forwarding cannot be implemented in level 1, because data synchronisation in liquid applications requires real-time notifications.

Level 2 - Client-Server Push - Similarly to the client-server pull level, clients are still the origin of the interaction with the server. However in level 2, clients open a two-way communication channel. In this level the server is allowed to propagate data and events to the connected clients immediately, meaning that it is possible to efficiently create real-time Web applications. The standard Web protocol used for implementing client-server push is WebSocket. With WebSocket it is possible to implement the liquid clone primitive since data synchronisation can happen in real-time. Liquid Web applications whose goal is to implement all possible liquid UX primitives need to consider at least a level 2 communication channel in the design of their architectures.

Level 3 - Peer-to-Peer - With the advent of the WebRTC protocol it is now possible to have peer-to-peer (P2P) communication among Web browsers. Architectures implementing level 3 communication channels still rely on the HTTP and WebSocket protocols for peer discovery purposes. Level 3 communication channels allow to lower the latency between clients, by potentially decreasing the number of hops in the communication, instead of propagating data relying on the server (client → server → client), in the best case it is possible to communicate directly between clients (client → client).

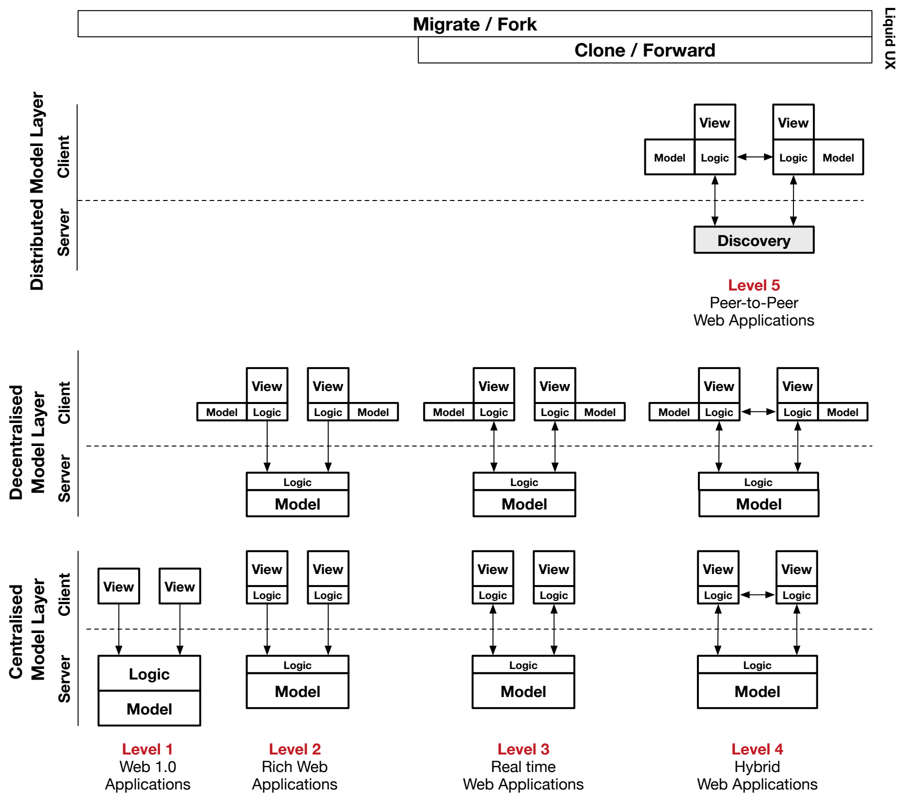

Fig. 4: Maturity model for Web architectures for centralised, decentralised and distributed model layer deployments. The controller layer is labeled as *Logic*.

4 Maturity Model

Figure 4 shows the maturity model of liquid Web applications determined by combining the deployment configuration of their MVC layers across the server-side and client-side with the choice of the communication channels established between them. We identify five levels: 1. Web 1.0 Applications 2. Rich Web Applications 3. Real-time Web Applications 4. Hybrid Web Applications 5. Peer-to-peer Web Applications

The diagram also shows in which level is possible to use the *migrate/fork* and *clone/forward* liquid user experience primitives. Migrate and fork are possible in all levels, while clone and forward can only be achieved starting from level three.

Table 1: Maturity model: architectural configurations and quality attributes.

	Configuration			**Quality Attributes**				
	Deployment		**Channel**	**Latency**	**Liquid UX**		**View**	**Privacy**
Level	Model	Logic		Hops	Migrate Fork	Clone Forward	Adaptation	Model
1	Centralized	Ultra-Thin	Pull	2	A	✗	Static	✗
2c	Centralized	Thin	Pull	2	A	✗	Responsive	✗
2d	Decentralized	Thin	Pull	2	A	✗	Responsive	✗
3c	Centralized	Thin	Push	2	A, S	✓	Complementary	✗
3d	Decentralized	Thin	Push	2	A, S	✓	Complementary	✗
4c	Centralized	Thin	P2P	1 or 2	A, S	✓	Complementary	✗
4d	Decentralized	Thin	P2P	1 or 2	A, S	✓	Complementary	✓*
5	Distributed	Thick	P2P	1	S	✓	Complementary	✓

Table 1 summarizes the different configurations per level as a combination of the facets explained in section 3. The table also describes how the configurations affect the following quality attributes:

Latency or the proximity between two clients on the network: • *2* hops means that whenever clients communicate to perform a liquid user experience primitive, the communication relies on an intermediary Web server (client $\rightarrow$ server $\rightarrow$ client); • *1* hop means that clients – in the best case – can communicate directly with each other.

Liquid UX (primitive) Migrate/Fork can occur asynchronously (A) or synchronously (S). **Asynchronous** migration and fork of an application happens between two clients that cannot directly push the migrated state and logic to the target client, but they have to be stored in a central storage first; **Synchronous** migration and fork of an application can be implemented in systems in which clients can push migrated or forked state and logic to other clients without the need to store it in a central storage. **Clone/Forward** indicates in which configurations the liquid clone and forward operations are possible ✓or not possible ✗.

View Adaptation describes which level of the view layer adaptability is possible to achieve in all configurations: *Static* means that the view does not dynamically adapt to the client hardware device capabilities, but it is displayed exactly as it was determined by the server; *Responsive* means that the view locally adapts to the client. *Complementary* means that it is possible to manually or automatically adapt the view to *set of* heterogeneous devices connected to the Web application.

Privacy describes if the users have control on their data by ensuring that it is exclusively stored in devices they own or trust: ✓ means that users are in control of their own data, ✗ means that the data is stored in untrusted devices, ✓* means that the data is stored in trusted devices, but is exchanged with or relayed across untrusted devices (e.g., a Web server running in the Cloud).

4.1 Level 0 - Solid Applications

All layers of a solid (or monolitic) application are deployed on the same machine, typically a standalone personal computing device, or a server to which multiple dumb terminal devices are attached. This architecture configuration predates the Web as intra-layer communication does not go through any Web protocol, but only happens locally within the same host using, e.g., local procedure calls or shared memory buffers.

Liquid migration can be achieved by the mean of input/output virtualisation of the clients, e.g. multiple users can access the operating system installed in the server by different screens. This architecture allows users to save their data on the server and access it from any screen. From the user perspective, switching terminal device amounts to successfully migrating their work from one screen to another. The *virtualised* client is therefore ultra-thin, where the view layer running on the server forwards the user interface input/output events and commands to the terminals connected to it.

The concept of the first Sun Ray [28] designed in 1997 can be considered as an example level 0, solid application architecture. The concept is designed to take advantage of stateless network computers whereby users authenticating with smart cards were instantaneously taken to their virtual desktops and could access their applications and data centrally managed on the server from anywhere.

4.2 Level 1 - Web 1.0 Applications

Level 1 Applications can be seen as the first generation of Web applications [4] built using the HTTP protocol. The logic and model layer are deployed on the server-side, while the view layers run client-side on Web browsers. In this level the content provided by the Web servers is static and cannot be changed by the clients. Web browsers retrieve content (e.g. Web pages written in HTML) by sending HTTP requests to Web servers addressed by URIs. Browsers display resources as they were sent from the server. The view layer is completely static, since there is no definition of a technology able to adapt the retrieved resources to different client rendering capabilities. At that time CSS media queries did not yet exist while the Extensible Stylesheet Language (XSL) did not provide any markup to adapt the content of a Web page to the device displaying it.

Level 1 supports asynchronous liquid migration by uploading resources to the server and then using their Uniform Resource Locator (URL) to retrieve the resource from another device. In this level cloning a liquid Web application is challenging to achieve, because data synchronisation does not happen in real-time, as clients can only resort to continuous HTTP polling. Likewise, migration in level 1 does not happen in real-time and requires the exchange or agreement on the URL addressing any resource that needs to be migrated between clients, which do not need to be available and connected at the same time.

This is the most basic architecture for implementing liquid applications that only need the liquid migration primitive, it does not provide any kind of view adaptation and cannot ensure data privacy (unless the Web server is owned and operated by the same organization owning the client devices). Synchronization between multiple clients can be achieved only by manually refreshing the Web page after sharing URLs via out-of-band channels, which does not fit with the real-time expectations of the liquid user experience.

4.3 Level 2 - Rich Web Applications

In Level 2 we consider rich Web applications [7] in which the controller layer is deployed both in the server and client-side. Level 2 architectures are the first ones able to have responsive views because the portion of the controller on the client-side can compute different views and do so based on the underlying hardware capabilities. Liquid migration is possible, but, like in level 1, shared URLs are needed to address and retrieve the resources representing the state to be shared among clients, since there is not a direct communication channel between clients. More in detail, after discovering the identifier of the resource being migrated, the Web application on the browser is manually refreshed to ensure the consistency of the displayed information with the model of the Web application. Again, cloning is hindered by the lack of real-time communication between clients attempting to immediately synchronize. The distance between two clients is always equal to two hops.

Depending on how the model layer is deployed on the clients we distinguish two different level 2 configurations: Level 2c - **centralised -** the model is deployed only in the server-side; Level 2d - **decentralised -** part of the model is stored in the client-side, in traditional Web applications it takes the form of *Cookies* or cached data, in modern rich Web application it takes various forms (e.g., local storage, service workers, WebSQL databases) as described in section 3. Users of a liquid rich Web application can store their confidential data in their devices, nevertheless during a migration of the liquid Web application its state has to be transferred via the server, which may not be always owned or trusted by the user of the Web application.

The liquid user experience in this level is similar to the one in level 1, with the addition of support for a responsive view and the option of storing parts of the model locally on the client.

Example Level 2 Frameworks CrowdAdapt [26] is a centralised level 2 framework for creating responsive Web pages. Web pages created with CrowdAdapt allow users to change the layout of the Web page as they desire and thereafter migrate their creations on other devices. In CrowdAdapt the controller layer provides the editing functionality and the automatic detection of the hardware specification of the device running the client. The users are able to choose between the layouts created by all the users of the Web application that better fit their needs.

PageTailor [5] is a decentralised level 2 framework with concepts similar to CrowdAdapt. Users can change the layout of Web pages using PageTailor and then reuse these layouts on subsequent visits. In this case layouts are not shared between multiple users as in CrowdAdapt.

4.4 Level 3 - Real-time Web Applications

The deployment of the view and controller layers are the same as in level 2, however level 3 applications have access to client-server push communication channels. This make it possible for Web applications to synchronise data among clients and notify connections of new clients and detect disconnections of old devices. The liquid clone operation is implementable in level 3 because data can be synchronised in real-time between simultaneously connected devices. The awareness of the connected clients to the Web application allows to distribute the view layer among them. The complementary view implementable in level 3 Web applications increases the quality of the liquid user experience. Liquid migration, liquid cloning and complementary view control can happen at different granularity levels: **application level -** the Web application is monolithic and all devices receive all the assets and model of the whole application. Upon migration or cloning the new clients have a perfect copy of the whole application whose state is kept synchronised between them. Complementary view adaptation in this case can be implemented through *Web clipping* by concealing part of a view on all but one device. **component level -** in component-based Web applications clients receive only portions of the whole application. Liquid migration and cloning can be done at component level, thus moving and keeping synchronized only part of the application. In this granularity level complementary view development does not need *clipping*, because clients move or receive only the portions of the application they need and do not have to locally hide the components displayed on other devices.

The **decentralised** configuration of level 3 allows partial privacy on the data created by the users, as it can be stored only on trusted devices. During liquid migration or cloning on simultaneously connected devices there is no need to store any information on the server. However data sent between clients is still relayed through the WebSocket channels on the server, meaning that such data must be encrypted in order to ensure privacy. In the case of liquid migration on devices which are not simultaneously connected to the Web application, the entire model has to pass through the Web server regardless.

Example Level 3 Frameworks Smart Composition [22] is a centralised level 3 framework that allow the creation of component-based (called widgets) multi-screen Web applications. By using a central *cross-device communication service* the infrastructure created by SmartComposition is able to compose distributed view layers among devices and keep the various components building the application synchronised.

Panelrama [38] is a centralised level 3 framework used to create distributed user interfaces using the concept of *panels*, Javascript objects defining pieces of user interface and logic. Panelrama provides an API to migrate and clone panels between devices and automatically create the complementary distribution of the view layer among the connected devices.

DireWolf [21] is a decentralised level 3 framework used to create multi-device mashups Web applications. Clients are aware of the connected devices in the application and can migrate widget-like components to any target device. Dire-Wolf offers the possibility to manage the device ownership, device information and specification, the widget state, and the application state of the whole application through its clients.

Liquid.js for DOM [35] is a decentralised level 3 framework based on *React.js* for component-based Web applications. By synchronising virtual DOMs between devices it is able to migrate and clone logic and model layers among the connected clients. It does not offer automatic cross-device complementary views. There also exists a level 4 Hybrid version of Liquid.js for DOM offering peer-to-peer data synchronisation between clients.

Bellucci *et al.* [3], Frosini *et al.* [11], and Raposo *et al.* [29] propose similar frameworks in which is possible to distribute and synchronise the view layer of the application on all connected devices.

4.5 Level 4 - Hybrid Web Application

Level 4 augments level 3 with the ability for clients to communicate directly with each other through P2P channels. The logic layer deployed in the client-side can send messages to other clients either directly with a single *hop* or through the server with two *hops*. Connected clients can send any kind of data between each other, including the entire assets of the application. Similarly to level 3, it is possible to have both asynchronous and synchronous migration and cloning operations among connected clients. Through the peer-to-peer channels decentralised hybrid Web applications can send confidential data directly among trusted clients without relaying any message through the server, ensuring privacy if confidential data does not need to be stored on the server.

Example Level 4 Frameworks XD-MVC [19] is a decentralised level 4 framework for creating cross-device interfaces and automatic complementary adaptation views applications. XD-MVC implements migration at the application level and takes advantage of clipping off parts of the view layer in order to simulate migration between devices. Views can be annotated with rules about how they are expected to adapt to the set of connected devices. Given these rules the view is able to dynamically and automatically adapt to set of heterogeneous devices when a new client connects or disconnects.

PolyChrome [2] is a centralised level 4 framework for creating co-browsing applications with collaborative views spanning on multiple screens deployed on multiple devices. PolyChrome complementary view adaptation supports *stitching*, *replication*, *nesting*, and *overloading* layouts. Data synchronisation happens both through peer-to-peer and WebSocket channels. The framework creates components out of a legacy applications in order to be able to make a view span on multiple devices.

4.6 Level 5 - Peer-to-peer Web Applications

Peer-to-peer [31] Web architectures are at the highest level of the maturity model, the only one providing all quality attributes expected from liquid Web applications. Peer-to-peer Web applications allows connected clients to communicate with each other directly. When peers are linked with a fully-connected mesh, this amounts to the best case scenario with a latency of 1 hop. Indeed other topologies are possible, like rings, in which N connected clients are up to $N/2$ hops away, or stars, in which the hops number vary between 1 and 2, depending on which peers are communicating.

Level 5 applications allow synchronous migration, since connected clients can push the model and logic through the full-duplex peer-to-peer channel created with WebRTC at any time. Since there is no longer a central server available at all times asynchronous migration is not possible. Instead clients need to be online simultaneously in order to proceed with the migration. Since clients sense and propagate their availability across the peer to peer network, it is possible to have complementary view adaptation.

Data privacy is ensured in peer-to-peer Web applications because users are in full control of all devices storing and processing their data. Data is never stored in any server or Cloud storage platform. Also, data migrated or synchronised with another device is never sent through a Web server.

Level 5 Web applications allow strong mobility with direct model and logic transfer and synchronisation between peers, however this requires to a suitable discovery method. WebRTC, for example, uses a signaling server to initiate and establish the connection between clients. Clients first connect to the signaling server and then receive information on how they can join the rest of the peers. Once the topology is created and peers are connected, they are free to communicate among themselves and the signaling server is no longer involved.

Example Level 5 Frameworks Liquid.js for Polymer [12] is a level 5 peer-to-peer framework which allows the creation of distributed component-based Web application built on top of the Polymer framework. Users instantiate any component provided by the Web application on their devices and share them directly with other users. If a peer does not own the assets of the component being sent to it, the peer will also receive the model of the component that is going to be migrated. Liquid.js allows to define strategies for creating different peer topologies. Currently Liquid.js for Polymer does not support automatic complementary view adaptation. However developers can build their own layout adaptations by using the API provided by the framework.

5 Related Work

Liquid software is named after a metaphor: as liquids adapt to the shape of their containers, *Liquid software* seamlessly adapts to the set of devices it is allowed to run on. In the Liquid Software Manifesto [34] Taivalsaari *et al.* make the case for liquid software, envisioning that the Web (thanks to its ubiquitous support) provides the most suitable platform for ensuring liquid applications can flow across heterogeneous devices.

The design space for liquid software is analysed in [13,14] where we defined twelve dimensions for designing liquid software and position existing technologies developed within the last two decades within the design space. In this paper we focus on the deployment configuration of Web applications designed according to the Model-View-Controller pattern and discuss more in detail how the configuration impacts the liquid Web application quality attributes.

Other fields close to liquid software concept are also working towards the definition and evaluation of technologies and architectures of frameworks which could be used to create applications able to seamlessly flow between multiple devices. Esenther [10] builds in 2002 a framework for real-time collaborative co-browser applications. Opera Unite (now discontinued) [27] was a Web browser extension allowing users to host social Web applications (e.g., photo sharing, social wall) on their Web browser. The goal of these efforts was to enable safe social networking whereby personal data would be exchanged directly between trusted devices.

Cross-device interfaces [18] study how to design collaborative environments spanning across multiple Web-connected devices. In the distributed user interfaces field, Santosa *et al.* [30] make a field study on the impact in the real world of the use of technologies enabling cross-device interactions. Given the responses of experts in the field, they collect and compare nine existing cloud-based data management software enabling cross-device collaboration between users.

The survey proposed by Elmqvist [9] discusses the state of the art of distributed user interfaces in the human computer interaction research area. Elmqvist summarizes how to achieve migration of the user interface and redirection of I/O of a device. The concept of forwarding used in this paper is similar to the concept of redirection used in the survey.

6 Conclusions

In this paper we described the maturity model of liquid Web architectures and provide examples of emerging frameworks and technologies across all five levels. Since the term *liquid software* was introduced 21 years ago [17], we recognise that there has been an evolution in how software architectures are designed to bring seamless application mobility across multiple devices.

In the context of Web applications, the maturity model presented in this paper includes: 1. Web 1.0 applications; 2. Rich Web applications; 3. Real-time Web applications; 4. Hybrid Web applications; 5. Peer-to-peer Web applications. Each level's architectural configuration impacts the possible liquid user experience primitives. Most of the existing liquid Web application development frameworks [2,3,5,11,12,19,21,22,26,28,29,35,38] we surveyed are categorised by a level 3 (real-time) architecture, however we emphasize that higher level architectures are possible and they should be considered to deliver all the quality attributes that one would expect from a liquid Web application, in particular data privacy (with decentralized configurations) and a reduced latency between devices that do not need to communicate with or through a remote Web server all the time. We acknowledge that real-world liquid applications with multiple client implementations may span multiple levels in the maturity model. For the sake of simplicity and clarity we described the main five levels in the maturity model instead of all possible combinations thereof.

The choice of level and configuration should be implemented or which framework to use are important architectural decisions. Upgrading an architecture from a lower level to a higher one, or downgrading to lower levels fundamentally impact the design of the Web application and are are likely to result in significant development costs. Still, over the history of the Web, application architectures have been gradually and steadily shifting towards the higher levels of the maturity model presented in this paper.

7 Future work

As the number of devices connected to the Web and the average number of devices owned by one user increases [16], more frameworks will appear positioned across all levels of the maturity model targeting the creation of liquid Web application. An evaluation of the presented and future frameworks in terms of performance, scalability, and usability would allow developers to assess which framework is more suitable for executing liquid primitives in the sequential and parallel scenarios.

HTML5 standards are quickly evolving every year and new specification drafts are already defining new technologies that may be used to further extend and improve the liquid user experience provided by liquid applications. While we described five levels in our maturity model, we do not exclude that in the future higher levels will appear. For example, emerging technologies like Web Bluetooth (currently not yet a W3C standard) [37] aims to bring Bluetooth support in Web browsers which may be used to define a new maturity level in which there is no longer a need for a central server in order to perform client discovery and device pairing.

We based our description on the MVC design pattern adopted by traditional Web applications, however in the future it may become necessary to revisit the fundamental architectural abstraction and design principles of Web applications and study their interplay with a programmable world [33] of billions of heterogeneous interconnected devices.

Acknowledgements This work is partially supported by the SNF with the ”Fundamentals of Parallel Programming for PaaS Clouds” project (Nr. 153560).

References

1. Abbott, M.L., Fisher, M.T.: The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Pearson Education (2009)

2. Badam, S.K., Elmqvist, N.: Polychrome: A cross-device framework for collaborative web visualization. In: Proc. of the Ninth ACM International Conference on Interactive Tabletops and Surfaces. pp. 109–118. ACM (2014)
3. Bellucci, F., Ghiani, G., Paternò, F., Santoro, C.: Engineering Javascript state persistence of web applications migrating across multiple devices. In: Proc. of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems. pp. 105–110. ACM (2011)
4. Berners-Lee, T., Fischetti, M., Foreword By-Dertouzos, M.L.: Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor. HarperInformation (2000)
5. Bila, N., Ronda, T., Mohomed, I., Truong, K.N., de Lara, E.: Pagetailor: reusable end-user customization for the mobile web. In: Proc. of the 5th international conference on Mobile systems, applications and services. pp. 16–29. ACM (2007)
6. Brusilovsky, P., Maybury, M.T.: From adaptive hypermedia to the adaptive web. Communications of the ACM 45(5), 30–35 (2002)
7. Casteleyn, S., Garrig'os, I., Maz'on, J.N.: Ten years of rich internet applications: A systematic mapping study, and beyond. ACM Transactions on the Web (TWEB) 8(3), 18 (2014)
8. DB-Engines: DB-Engines ranking. `http://db-engines.com/en/ranking` (2017)
9. Elmqvist, N.: Distributed user interfaces: State of the art. In: Distributed User Interfaces, pp. 1–12. Springer (2011)
10. Esenther, A.W.: Instant co-browsing: Lightweight real-time collaborative web browsing. In: Proc. of WWW (2002)
11. Frosini, L., Manca, M., Paternò, F.: A framework for the development of distributed interactive applications. In: Proc. of the 5th ACM SIGCHI symposium on Engineering interactive computing systems. pp. 249–254. ACM (2013)
12. Gallidabino, A., Pautasso, C.: Deploying stateful web components on multiple devices with liquid.js for Polymer. In: Proc. of CBSE. pp. 85–90. IEEE (2016)
13. Gallidabino, A., Pautasso, C., Ilvonen, V., Mikkonen, T., Systä, K., Voutilainen, J.P., Taivalsaari, A.: On the architecture of liquid software: technology alternatives and design space. In: Proc. of WICSA. pp. 122–127. IEEE (2016)
14. Gallidabino, A., Pautasso, C., Ilvonen, V., Mikkonen, T., Systä, K., Voutilainen, J.P., Taivalsaari, A.: Architecting liquid software. Journal of Web Engineering (2017)
15. Ghiani, G., Paternò, F., Santoro, C.: On-demand cross-device interface components migration. In: Proc. of the 12th international conference on Human computer interaction with mobile devices and services. pp. 299–308. ACM (2010)
16. Google: The connected consumer. `http://www.google.com.sg/publicdata/explore?ds=dg8d1eetcqsb1_` (2015)
17. Hartman, J., Manber, U., Peterson, L., Proebsting, T.: Liquid software: A new paradigm for networked systems. Tech. Rep. 96-11, University of Arizona (1996)
18. Husmann, M., Marcacci Rossi, N., Norrie, M.C.: Usage analysis of cross-device web applications. In: Proc. 5th ACM Intl. Symposium on Pervasive Displays. pp. 212–219. ACM (2016)
19. Husmann, M., Norrie, M.C.: XD-MVC: Support for cross-device development. In: 1st Intl. Workshop on Interacting with Multi-Device Ecologies in the Wild (Cross-Surface 2015). ETH Zürich, Switzerland, Zürich (2015)
20. Jazayeri, M.: Some trends in web application development. In: Future of Software Engineering, 2007. FOSE'07. pp. 199–213. IEEE (2007)

21. Kovachev, D., Renzel, D., Nicolaescu, P., Klamma, R.: Direwolf - distributing and migrating user interfaces for widget-based web applications. In: International Conference on Web Engineering. pp. 99–113. Springer (2013)
22. Krug, M., Wiedemann, F., Gaedke, M.: Smartcomposition: a component-based approach for creating multi-screen mashups. In: International Conference on Web Engineering. pp. 236–253. Springer (2014)
23. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
24. Marcotte, E.: Responsive Web Design. Editions Eyrolles (2011)
25. Mikkonen, T., Systä, K., Pautasso, C.: Towards liquid web applications. In: Proc. of ICWE, pp. 134–143. Springer (2015)
26. Nebeling, M., Speicher, M., Norrie, M.C.: CrowdAdapt: enabling crowdsourced web page adaptation for individual viewing conditions and preferences. In: Proc. of the 5th ACM SIGCHI symposium on Engineering interactive computing systems. pp. 23–32. ACM (2013)
27. Opera: Opera Unite reinvents the Web (2009), `http://www.operasoftware.com/press/releases/general/opera-unite-reinvents-the-web`
28. Oracle: Sun Ray products (2016), `http://www.oracle.com/technetwork/server-storage/sunrayproducts/overview/index.html`
29. Raposo, M., Delgado, J.: Empowering the web user with a browserver. In: Proc. of the International Conference on ENTERprise Information Systems. pp. 71–80. Springer (2010)
30. Santosa, S., Wigdor, D.: A field study of multi-device workflows in distributed workspaces. In: Proc. of the 2013 ACM international joint conference on Pervasive and ubiquitous computing. pp. 63–72. ACM (2013)
31. Schollmeier, R.: A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: Proc. of the First International Conference on Peer-to-Peer Computing. pp. 101–102 (2001)
32. Sivasubramanian, S., Pierre, G., Van Steen, M., Alonso, G.: Analysis of caching and replication strategies for web applications. IEEE Internet Computing 11(1), 60–66 (2007)
33. Taivalsaari, A., Mikkonen, T.: A roadmap to the programmable world: Software challenges in the IoT era. IEEE Software 34(1), 72–80 (Jan/Feb 2017)
34. Taivalsaari, A., Mikkonen, T., Systa, K.: Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In: Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual. pp. 338–343. IEEE (2014)
35. Voutilainen, J.P., Mikkonen, T., Systä, K.: Synchronizing application state using virtual DOM trees. In: Proc. of the 1st International Workshop on Liquid Software (2016)
36. Wallis, M., Henskens, F., Hannaford, M.: A distributed content storage model for web applications. In: Proc. of Evolving Internet (INTERNET), 2010 Second International Conference on. pp. 98–106. IEEE (2010)
37. Web Bluetooth Community Group: Web bluetooth. `https://webbluetoothcg.github.io/web-bluetooth/` (2017)
38. Yang, J., Wigdor, D.: Panelrama: enabling easy specification of cross-device web applications. In: Proc. of the 32nd annual ACM conference on Human factors in computing systems. pp. 2783–2792. ACM (2014)