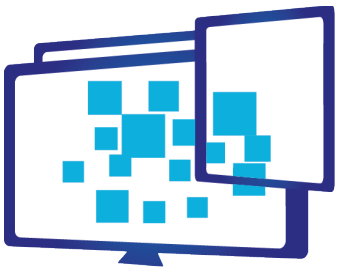

MediaScape

Dynamic Media Service Creation, Adaptation and Publishing on Every Device

Instrument: Collaborative Project (STREP)
Call Identifier: FP7-ICT2013-10
Grant Agreement: 610404

WP4 - Synchronization

Deliverable Number	D4.4
Title	MediaScape multi-device synchronization services
Version	1.0
Date	23rd May 2016
Status	FINAL

Public

Project Partners: VIC, IRT, NEC, BBC, W3C, NOR, BR

Contributors: **NOR, IRT, W3C, VIC**

Document Control

Version	Date	Author	Modifications
1.0	18. Mar. 2016	Njål Borch (NOR)	Libre Office document from a Google Docs with contributions of different partners
1.0	18. Mar. 2016	Mikel Zorrilla (VIC)	Peer Review
1.0	23. May 2016	Ingar Arntzen (NOR)	Formatting and minor changes
1.0	23. May 2016	Mikel Zorrilla (VIC)	Final changes

Table of Contents

1. Executive Summary

One of the biggest challenges for multi-device Web applications is the management of objects and their states across multiple devices, as elements and functionality move from one device to another, or exist on multiple devices simultaneously. This has vastly increased complexity for developers building multi-device applications capable of running in heterogeneous and dynamic network conditions while providing a consistent user experience.

In Work Package 4, we have focused on creating generic mechanisms and technologies to provide powerful yet simple ways to program multi-device applications for the Internet. The services and tools created allow programmers to focus on a single, coherent application even if it will run on a variety of devices in different networks.

In this deliverable, we present the Shared State model, with services and tools to support a large variety of applications. The Work Package is divided into three tasks, 4.1 targets data sharing, 4.2 targets a multi-device runtime environment and 4.3 targets timing issues.

Task 4.1 seeks to provide consistent synchronization of data in multi-device applications, allowing developers an easy way to store, retrieve and share runtime data between all devices in an application.

Task 4.2 builds on the consistent data sharing service of Task 4.1, adding contextual information, device capabilities, sensor bindings etc., creating a Shared Context for applications. This allows programmers an almost trivial way to discover the participating devices and what they can do, and use the devices as they wish. Simple examples include reacting to phones being shaken, monitoring microphones for applause or simply determining the screen resolutions and aspect ratios of all participating devices.

Task 4.3 provides powerful abstractions and tools to handle timing related issues. Shared Motion provides a kind of online, remote controllable clock for the application. Shared Motions are shareable between multiple devices or multiple users, or just for a single user changing between devices (like bookmarks). To ease application development, a Sequencer has been created as well, making it very easy to trigger processing of timed data at the exact correct time. Examples can be subtitles, timing screens for sport, transitions between elements, videos in a playlist etc. Finally, Task 4.3 also provides a MediaSync library to hook existing HTML Media Elements to Shared Motions, effectively making them remote controlled and integrated into the MediaScape experience.

All the outcome libraries of Work Package 4 have been used in the final MediaScape prototypes. For example, in the Elections prototype the Shared State model was used to create a Shared Context for the different views of a single user. Shared Motion, MediaSync and the Sequencer was used for the synchronization of audio, video and timed data. The Work Package has also been highly active in W3C standardization efforts, in particular of the Timing Object, which is very closely related to the Shared Motions. The timing object standardization effort was presented at NAB 2016 by MediaScape project partners, and the interest gained by this proposal is a testament to the success of MediaScape.

2. Introduction

Multi-device Web applications involve managing states, elements and functionality across different browsers context hosted (possibly) by different devices. Providing consistent user experiences across devices has been a huge challenge for developers, in an environment characterized by heterogeneity on all levels; from low-level network conditions and CPU power, to higher-lever rendering and input capabilities. More often than not, this has resulted in increased complexity for application developers. With regard to these issues, this work package makes use of generic mechanisms and technologies to provide an easy to use programming model for multi-device applications. It provides multi-device synchronisation for objects at different levels, e.g., from generic timed data at the lowest level to user interface state at the highest level, therefore largely simplifying the development of multi-device Web applications.

In this work package, a suite of synchronisation and state sharing mechanisms is created to provide efficient lightweight consistency and latency-tolerant synchronisation. The work package addresses both dynamicity and heterogeneity of networks and devices as well as simultaneous user interactions from multiple devices/multiple users in a multi-device environment. This work package provides architecture, abstracted APIs and reference implementations for synchronisation of time-referenced data, application context, and shared motions in multi-device, Web-based applications.

3. Requirements

MediaScape has defined a few central scenarios and a wider set of functional use cases. Based on these use cases technical requirements have been derived. The following table lists requirements relating to synchronisation and WP4.

Req 4.1	UC1R3, UC12R3	Device Notification	• Devices receive notifications of state changes
Req 4.2	UC1R4, UC2R3, UC4R2, UC9R2, UC10R3, UC11R3, UC13R2, UC21R2, UC22R2	Timeline based synchronisation	• extra material synchronised with programme timeline • migrate synchronised extra material • multi-device synchronised extra material • record user interaction according to programme timeline
Req 4.3	UC3R1, UC8R2	Media Synchronisation	• Synchronisation of audio and video • Different sources, different agents
Req 4.4		Shared (remote) Media Control	• Symmetric media control (e.g., group) • Asymmetric media control (e.g. broadcast)
Req 4.5	UC7R1, UC13R1, UC15R1	Pause/resume/seek	• Flexible media navigation • Switch from on-demand to live
Req 4.6	UC4R3	Mash-up, Extensibility (~~Injected Metadata~~)	• Expose media and timing resources • 3rd parties provide extended services

The above requirements fall broadly into three categories, data consistency (Req 4.1), time-sensitive synchronisation (Req 4.2-4.5) and modularisation (Req 4.6).

- Data consistency is addressed by the concept of shared state, which includes matters of distributed data consistency and propagation of state changes. See Shared State programming model (chapter 6) as well as task 4.1 Data (chapter 7) and task 4.2 Context (chapter 8).
- Time-sensitive synchronisation, including clock synchronisation, distributed media control and synchronisation of time-referenced media are all addressed by task 4.3 Timing (chapter Error: Reference source not found).
- Modularity follows as an implication of the shared state programming model (chapter 6) as well as isolation of shared motion as an independent resource (chapter Error: Reference source not found).

4. Synchronisation

The term “synchronisation” is widely used and is open to different interpretations.

One use is synchronising data between multiple devices, such as a phone and a computer sharing a common address book. In this case, the term synchronisation relates to propagation of state and consistency of the address book. Essentially, when altering the state of a shared object, the new state (or diffs) must propagate to all representations, across all devices. This is commonly labelled synchronisation, or data synchronisation. In MediaScape, we do **not** adopt this usage. Instead, we use the term **shared state/objects** for this purpose. The address book is regarded as a **shared object**, and change propagation and multi-device consistency are implied by definition.

Instead, MediaScape reserves the term synchronisation for matters that explicitly relate to time or timing. The core idea of time-based synchronisation is to make **data availability depend on computer clocks (predictable) rather than propagation delay (unpredictable)**. By associating state transitions with timestamps - or by defining the validity of object states in terms of time intervals, we can implement data abstractions with built-in support for precise time synchronisation, thereby simplifying timing challenges for programmers. In this model the availability of a shared clock is required, and synchronisation precision ultimately depend on the precision of this clock.

MediaScape explorse solutions for multi-device state sharing as well as state synchronisation (i.e. time-sensitive data delivery). For example, shared state solutions likely suffices for contextual data such as user presence or device presence. In contrast, whenever a companion app screams “GOAL!” 10 seconds before the goal appears on TV, the need for correctly timed data delivery is evident.

Though both shared state solutions and synchronisation are required in MediaScape, the research contribution of WP4 mainly relates to time-based synchronisation, as this topic has not been extensively researched within the context of multi-device Web applications.

5. Work Package Structure

Within WP4 we have identified three high-level challenges, represented as three independent tasks; **Data (Chapter 7)**, **Context (Chapter 8)** and **Timing (Chapter Error: Reference source not found)**.

Illustration 1: Main tasks Work Package 4

5.1. T4.1 Data

Task 4.1 provides generic mechanisms for data sharing and synchronisation in multi-device Web-applications:

- Data sharing: Solutions target a consistent view of the current state.
- Synchronisation: Data and data changes are associated with points or intervals in time, and synchronisation targets the availability of such time-referenced data consistent with time progression. This has been solved by wrapping a Sequencer from T4.3 on top of shared data.

5.2. T4.2 Context

Web-based multi-device applications depends on shared access to a common application context. This context represents the application itself, including for instance:

- agent presence
- device capabilities, device status
- roles
- etc.

T4.2 defines methods to capture, monitor and modify generic application context, while keeping it consistent across multiple devices. In this regard, T4.2 leverages mechanisms provided by T4.1. The application context allows programmers to create context-sensitive, adaptive applications with minimal effort.

5.3. T4.3 Timing

Task 4.3 provides generic solutions for a variety of challenges related to timing and time-based synchronisation. The task provides a programming model where time and timing aspects are explicit, making it easy for programmers to implement application specific timing requirements. In particular, T4.3 provides:

- a mechanism for precise clock sharing among connected devices
- a mechanism for time-synchronisation of continuous media (e.g., audio/video)
- a mechanism for time-synchronisation of non-continuous, linear media (e.g., subtitles, time-series, timed data etc.)

The mechanisms developed by T4.3 are instrumental in the development of time-based data synchronisation in T4.1.

6. Programming Model - Shared State

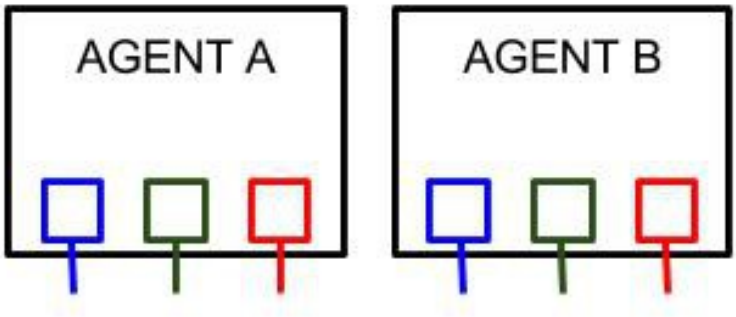

Illustration 2: Shared State programming model. Agents interact with local representations of shared resources.

MediaScape has chosen the shared state model as a programming model. Shared state means that **agents have access to the same data model, and are able to query and update the data, as well as receiving events when data is changed**. With this model, programmers do not see any explicit message passing, only local interaction with a local state object. State changes can be detected by querying the state object, or by means of an event listener. The above illustration shows two agents, A and B, and indicate how data (blue), context (green) and timing (red) are made available to agents -- locally, as shared state. If agent A writes something to data (blue), agent B will be able to detect this, and possibly perform application specific reactions.

The shared state programming model is not a radical suggestion, but rather the way modern multi-user applications are commonly built. For instance, the Facebook Graph API exposes a shared data model to programmers (through a RESTful API and a notification system). Still, we emphasise this programming model here to ensure that the MediaScape architecture is fully in line with the architectural principles of the Web and modern cloud services, as well as their associated programming models. In the following we outline some of the reasons why the shared state model is a natural fit for Web-based, multi-device applications.

The shared state model makes life easy for programmers, by providing a high level and simple abstraction. In particular, shared state hides complexities of the distributed environment, such as connection management, messages passing, RPC calls, data persistence and consistency.

Shared state also encourages a highly event based programming model, where application logic on the client essentially mediates between shared state and UI. For example, event listeners attached to the UI will essentially translate user interaction into operations on shared state. In the opposite direction, event listeners on shared state will translate state changes into UI operations.

Another benefit with the shared state model is that it decouples agents from each other. In order to collaborate, it is not necessary that agents communicate directly, or even know about the existence of each other. This is crucial on the Web, as browsers connect and disconnect as they please. In this environment, it makes sense for agents to operate independently towards a shared, dependable data model. In addition, Web-browsers have no state initially, and need to download all required resources on load (and reload). This too is a good match with the shared state model, as Web-browsers can always depend on shared state to include the complete (and up-to-date) set of required resources.

Shared state does not have to be monolithic. There is no requirement that all application data is collected in the same data model, or even in the same service. As indicated in the above illustration, an application may load multiple components that each independently implement the shared state abstraction. These components are likely to represent independent resources and export their own specialised APIs. For example, in the illustration above, the context (green) may be a standard component that all applications make use of, whereas the data

(blue) may be an entirely application-specific component. Again, this is parallel to how standard Web applications are built from a set of online resources/services. To further simplify the life of the programmer, cloud services are already offering generic shared state services, for instance, Google and Amazon offer cloud-hosted key-value stores and databases.

Finally, note that shared state is an **abstraction**, and it is open to different implementation, e.g., a centralized approach using back-end servers, or a decentralised approach based on P2P protocols, or even a combination of both. Local network communication may also be used to implement shared state, though in this case sharing will be limited to agents on the local network.

6.1. Architecture Design

The shared state programming model is open to multiple implementations. However, as MediaScape targets connected services, WP4 primarily focuses on an online, centralised architecture. As illustrated below, this implies an architecture where data (T4.1), context (T4.2) and timing (T4.3) are represented as online services. From the perspective of programmers, these online services are available locally through an associated JavaScript library, thereby implementing the shared state abstraction.

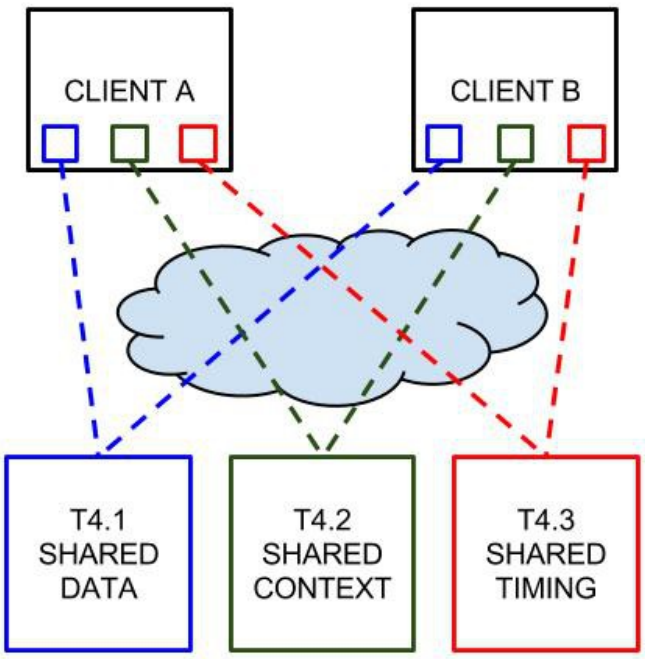

Illustration 3: MediaScape architecture: Resource sharing implemented as independent, online services.

In this illustration two clients individually connect to three online services, and stay connected to receive state updates from each of the services. The two clients are independent of each other, and may join and leave the application at any time. If the application requires clients to know about each other, this information must be part of shared state. In this case presence information may be available as part of the shared context.

Online services are independent too. There should be no server-side processes on behalf of any specific client or application. Instead, services should efficiently be processing well defined (e.g., RESTful) queries and updates. Services implementing general purpose data structures may be managing state on behalf of many applications. Services must enforce access control restrictions that applications define for objects in shared state.

However, despite the independence of clients and services, the two clients above will appear to be tightly integrated. This is because clients can always agree about application data, about the context of the application, as well as application specific timing aspects. Furthermore, by scrupulously adhering to the architectural

principles of the Web, the shared state architecture may offer the flexibility, extensibility and scalability that we expect from modern Web-based applications.

6.2. Shared State Protocol

In order to implement the shared state abstractions, clients and servers need to agree on an appropriate protocol. The main purpose of the this protocol is to allow the client to maintain a local proxy object consistent with a server-side object. Though details of such protocols may differ between services and data models, there is a common pattern:

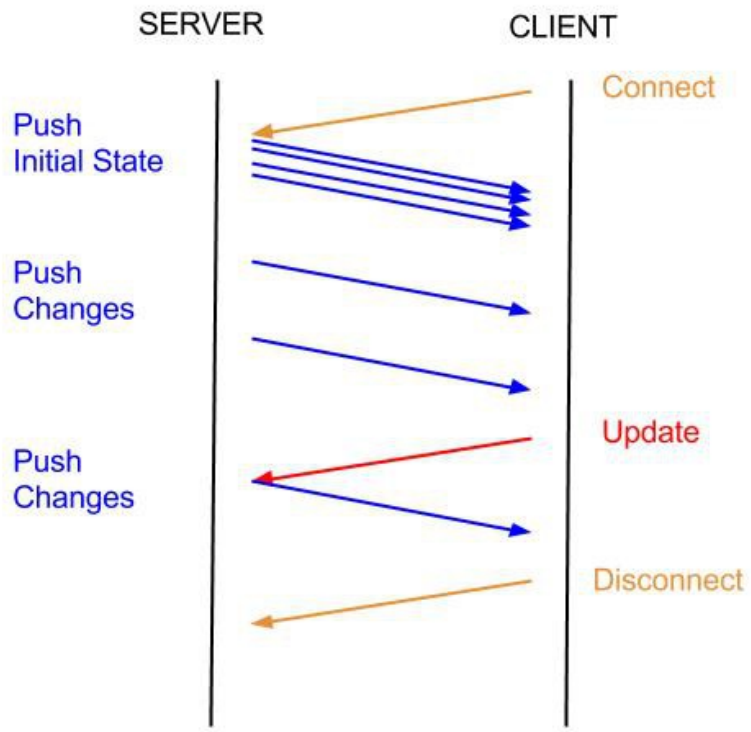

Illustration 4: Generic protocol for clients connecting to shared state services.

1. After connecting (ORANGE), the client receives the full state of the object (BLUE). In the perspective of the client, this is the initial state. The initial state may be communicated as a serialized copy of the entire server-side object, or as a series of operations that produces the correct state.
2. The server is responsible for notifying all connected clients whenever the state is changing (as long as the connection is open). In the above illustration there are two more (asynchronous) data transfers (BLUE) from server to client. These are caused by changes in the server-side object, possibly inflicted by other clients. Data messages may be modelled as new versions of altered state, or diffs. In any case, upon receipt the client will update the local proxy object and fire events to notify application code.
3. The client may update the state of the server-side object itself, by issuing an update request (RED). The illustration shows how this update triggers a new data transfer (BLUE) (to all connected clients) reflecting the effects of the update request. The update request does not take effect at the issuing client, until **after** the update request has been processed by the server and changes pushed back to the clients.

Essentially, this describes a variation of the master-slave replication pattern, where the server is the master and clients act as slaves maintaining replicas. WRITE (update) operations are processed only by the master, while READ (query) operations are processed exclusively by the slaves.

In this protocol description, messages are ideally exchanged over a single, reliable, duplex network connection. The connection is opened by the client and terminated by either server or client. Duplex communication may for instance be implemented using WebSockets, or (if WebSockets are not available) emulated through regular

synchronous HTTP GET in combination with long-polling (i.e., blocking HTTP GET). Alternatively, persistent HTTP connections allow dynamic updates to be sent asynchronously as (multipart) HTTP responses. Ajax [AJAX] is another common technique for keeping web pages up to date with dynamic server state, or posting changes without necessarily reloading the page.

6.3. Shared State Consistency

The main goal of shared state solutions is to let distributed agents agree on the current state, ideally at all times. Consistency models for distributed storage abstractions are typically discussed in terms of ordering of READ and WRITE operations, and their ordering as observed by different clients. The shared state abstraction indicates a model where READ operations are local, i.e., evaluated towards a local representation of shared state. In contrast, WRITE operations must be propagated across the network to have effect in the distributed setting.

In a centralised architecture it is easy to ensure global ordering of WRITE operations, simply by asserting that update operations are serialized by the shared state service. This way, we can ensure that all clients eventually observe the same sequence of WRITE operations. However, if a client issues a WRITE operation immediately followed by a READ operation, the READ operation will (likely) NOT reflect the effects of the WRITE.

This model is already pervasive on the Web. The non-blocking execution model of JavaScript effectively demands that blocking operations must be modelled with callbacks and readyStates. This makes explicit the distributed nature of the Web environment. The shared state model should support callbacks on update completion, and additionally Promises [PRO] as an alternative. This will be helpful in recognising failures. In any case, the shared state model encourages a programming pattern where code for issuing updates and reactions to updates (events) are separate concerns. Consequently, completion callbacks for updates are only required if there is specific actions to be done by the specific agent that issued the update.

In the distributed scenario there will always be clients communicating over slow networks. This means that at a given point in time, clients may disagree about the shared state simply because update notifications have not yet reached all clients. So, it is possible for clients to make decisions and issue new update operations based on an "out-dated" view of the state. This does not threaten the integrity of shared state (i.e. global ordering of WRITE operations). Still, if needed, it is possible to ward against this. For instance, a global update counter may be used to recognise and deny update out-dated requests, i.e. requests originating from agents with an out-dated version of the shared object. In many circumstances though this is not required, so our shared state solutions do not support this as default behavior.

Finally, time synchronisation offers statistical improvement by introducing delay, thus effectively reducing the variation in propagation delay (from the perspective of application code). This is potentially a trade-off against latency in user interaction. Note also that the availability of shared clocks (see chapter Error: Reference source not found) makes it possible to implement more sophisticated consistency models taking time-ordering into account.

7. Task 4.1 Data

Task Leader: IRT

Task 4.1 provides generic mechanisms for data sharing and synchronisation in multi-device Web-applications:

- **Data sharing**: Solutions provides a consistent view of the current state.
- **Synchronisation**: Data or data changes are associated with points or intervals in time, and synchronisation allows rendering of such time-referenced data consistent with time progression. This has been solved by wrapping a Sequencer from T4.3 on top of shared data.

Task 4.1 explores different implementations of these mechanisms. In the online case, data sharing mechanisms may be implemented using back-end servers or by means of direct client-to-client communication. The same options are available in circumstances where the multi-device applications are constrained to the local network. However, MediaScape targets online, connected services, so the main focus has been on an online, centralized architecture.

Furthermore, data sharing mechanisms are designed especially according to requirements of online Web-based multi-device applications. This does not preclude the usage in LAN settings, nor does it preclude the usage of LAN-specific functions in implementation. However, it does emphasise the importance of avoiding LAN-specific assumptions in the general design. Especially, with respect to latency, network properties, consistency, dynamicity, scale, and concurrency, assumptions must be valid in the WAN environment.

7.1. State Sharing

There already exist several hosted solutions that offer state sharing through common data structures such as key-value mappings, file system abstractions and databases. MediaScape relies on existing solutions as much as possible, and interfaces are adapted to match with the shared state model and MediaScape programming idioms.

7.2. Time-based Synchronisation

A core ambition for MediaScape was to develop a programming model that simplifies implementation of applications with strict timing requirements. Precise timing is generally important in multi-device applications, because issues with time-consistency directly affects the end user experience negatively. User interfaces that have timing issues may be interpreted as slow, confusing, or even make users falsely suspect that the application has failed (until it suddenly seems to recover). Timing issues may also lead to unintended spoilers, subverting the main purpose of the application. In contrast, tight and dependable timing supports the idea that multiple devices are natural parts of the same application, allowing end users to forget about technology and rather just enjoy the application.

In MediaScape we have approached this by using a Sequencer to ensure precisely timed delivery of data to the application. For example, when updating a shared object, the update operation could be associated with a delivery time. Even though the data is transferred to the clients before the delivery time, the delivery of the data to the presentation is correctly timed. In a multi-device Web application such a mechanism can be exploited directly to make a single event be reported consistently (in synchrony) by multiple devices.

In the context of linear media, media content often binds naturally to time intervals. For example, Illustration 5 shows a linear media presentation composed from different data types, including text, images and video. They are all referenced to a common timeline, and each media object is associated with a time interval. As playback-time progresses along this time-axis these time intervals defines validity of individual data objects.

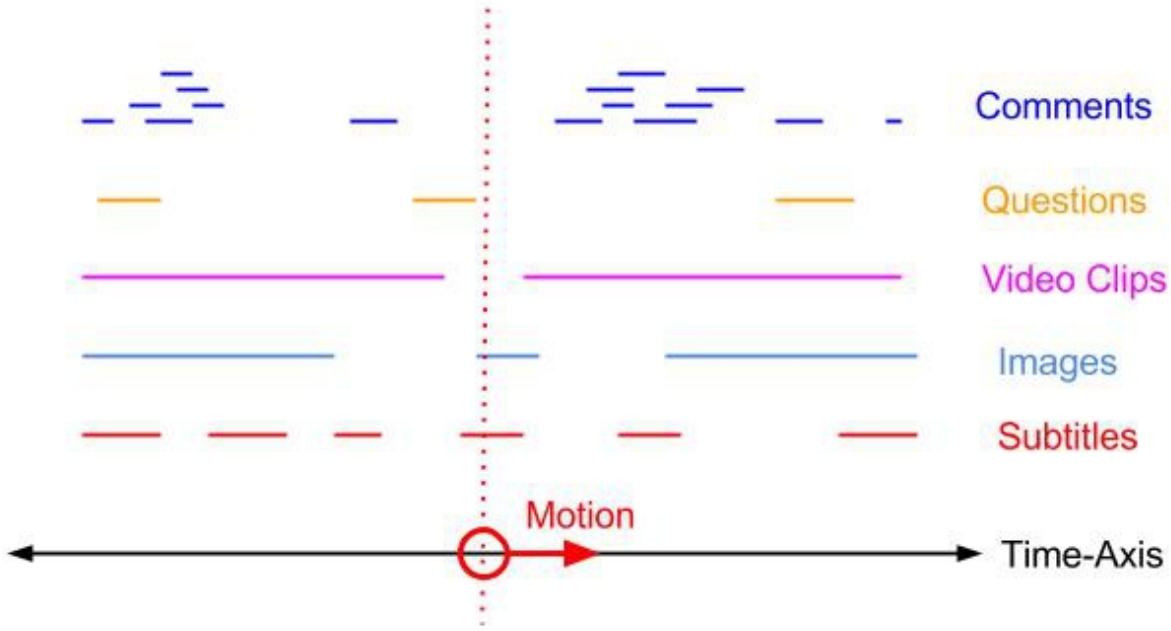

Illustration 5: Timed data mapped to time-axis. Sequencing of timed data based on motion along time-axis.

MediaScape have explored time-sensitive shared state abstractions, where objects are referenced to time by means of *points* and *intervals*. Time-sensitive shared state abstraction ensured that individual objects are

made available/unavailable based on time. For instance, consider the object “subtitle” from the illustration above. Querying this object repeatedly as time passes will return different subtitles - or no subtitle at all if queried “in-between” subtitles. Time-sensitive shared state solutionsalso allows time-referenced data to be dynamically modified, in terms of the object state as well as interval of time-validity.

Precise timing in distributed shared state abstractions ultimately depends on precisely synchronised clocks. Here, MediaScape relies on the timing mechanism provided by Task 4.3 Shared Motions9.2. This choice also informed the architecture of our time-sensitive shared state solutions. As a general design principle, precise timing should **not** depend on servers pushing data to clients **at the right time**. Instead, servers push objects if possible **ahead of time**, leaving it to clients to make objects available locally **at precisely the right time**. Furthermore, there is no requirement to use push-based solutions. Since clients are timing-aware, they can alternatively use pull-based solutions to fetch data ahead of time.

All of these requirements match nicely with the shared state protocol defined in chapter 6.

- In the shared data model objects are associated with points/intervals.
- Update operation allows either object value or point/interval to be changed
- Shared state provides agreement on the data as well as the timing, allowing clients to operate on shared data in a timing consistent manner.
-

7.3. Task Outcome

The research contribution of T4.1 mainly relates to consistent synchronisation of data in multi-device applications. The core idea was to include time-sensitivity into shared state solutions, thereby making it easy for developers to express application-specific, time-sensitive behavior. The proposed architecture is based on the assumption that clients are timing-aware. This assumption is supported by the timing mechanism in T4.3 Shared Motions . Implementation and specification of APIs in T4.1 was based on this high level architecture.

8. Task 4.2 Context

Task Leader (NOR)

T4.2 have designed and implemented mechanisms/solutions to capture, share, and maintain the states of application objects across multiple devices in a distributed environment with a number of connected devices. The solution includes:

- an agent context object representing the current state (high-level) of the agent based on analysis of sensors, system information and other locally available services.
- a shared application context, providing all agents with a consistent view of contextual information contributed by connected agents.

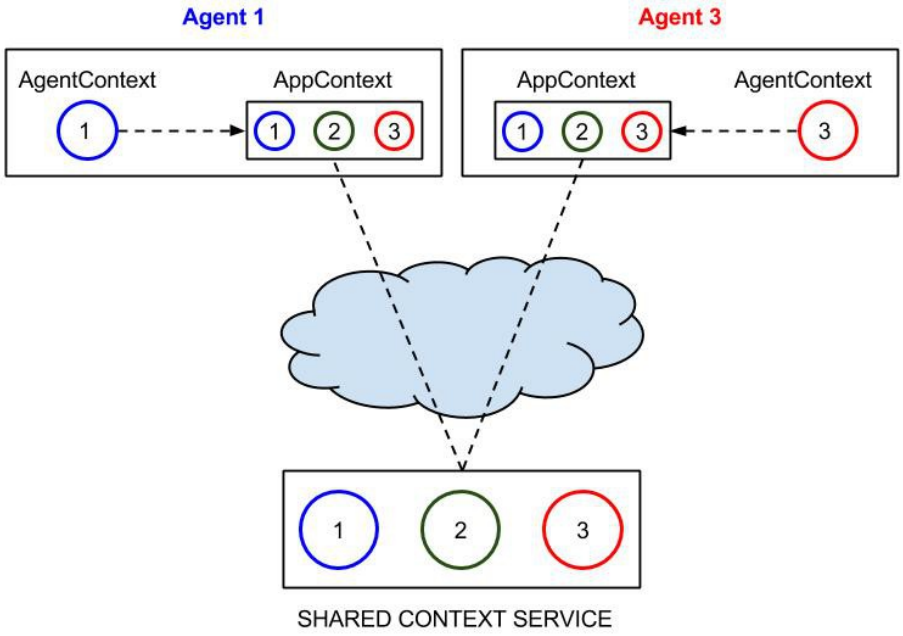

Illustration 6: Shared contex based on the shared state architecture.

Illustration 6 shows the difference between **Agent Context** (circle) and **App Context** (rectangle). **Agent 1** (**BLUE**) and **agent 3** (**RED**) each have their own agent context. This information is then made public through the *AppContext*. The *AppContext* provides representations for all participating agents, including **self**. (For Agent 1, *agentContext.self* points to the blue circle within the rectangular *AppContext*.)

When viewed from a practical angle, it is easy to appreciate the distinction between *AgentContext* and *AppContext.self*. For instance, consider the case where Agent1 wants to listen to samples from ***its own*** accelerometer. The trivial way to achieve this is to register an event handler on the *AgentContext*. This should provide high frequency samples with low latency. Alternatively, it is also possible to register the handler on *AppContext.self*. Since this involves propagation of samples over the Internet, it would likely give higher latency (over 1 RTT), and possibly lower-frequency (as it may be advisable to reduce the frequency for distributed propagation). On the other hand, the output would be consistent with the view provided to other agents.

The App Context provides a foundation for the Adaptation Engine of WP5, providing access to information necessary to consistently determine the capabilities and the roles each agent should have at any given time.

As T4.2 does not dictate particular data structures, it might be implemented by T4.1, or use existing key/value based cloud services as well as a presence service. This added flexibility makes T4.1 easier to integrate in existing applications, as existing backends can be re-used also for the application context.

As with all WP4 services, agents should respond to changes in the shared state, even when setting (requesting) the state change themselves. This ensures consistent application behaviour regardless of which agent initiated the state change.

8.1. Task Outcome

T4.2 have defined two logical components related to contextual information. The agent context is private to the particular agent, essentially a characterization of the execution environment. The agent may export (a subset of) this information to the application context, thereby sharing it with other agents within the same application. This high level architecture was the basis for specification of API's and implementation in T4.2.

9. Task 4.3 Timing

Task Leader: NOR

Task 4.3 provided generic solutions for a variety of challenges related to timing and time-based synchronisation. The overall goal was to provide a programming model where time and timing aspects are explicit, making it easy for programmers to implement application specific timing requirements. In particular, T4.3 provides:

- a mechanism for accurate clock sharing among connected devices
- a mechanism for synchronisation of continuous media (e.g., audio/video)
- a mechanism for synchronisation of non-continuous, linear media (e.g., subtitles, time-series, timed data etc.)

The mechanisms developed by T4.3 are instrumental in the development of time-based data synchronisation in T4.1.

9.1. Shared Clocks

Precise time-based synchronisation on the Web depends on agents (e.g., Web-browsers) having precisely synchronised clocks. Unfortunately, synchronised clocks is not a valid assumption in the current Web environment. NTP (Network Time Protocol) has been around for a long time and provides a practical, well-proven and widely deployed solution. However, mobile devices typically do not synchronise their clocks frequently, so millisecond precision can not generally be expected.

T4.3 provides synchronised clocks as a resource for application developers, in a similar fashion to data or application context. For example, an application may define a shared clock associated with linear media, such as video. Sharing such a video clock would allow multiple devices to maintain precise agreement about video offset, thus enabling collaborative viewing and time-sensitive secondary device applications. Similarly, broadcasters may define reference clocks for each broadcasted program. Currently they do so by including MPEG-TS timestamps in the DVB streams. However, as this clock is limited to clients actually consuming the stream, we additionally imagine more generic, content-independent clocks, usable for synchronised, Web-based secondary screen offerings etc.

9.2. Shared Motions

Shared Motion is a concept [MSV] offering a generic timing mechanism for the distributed environment, designed for but not limited to the Web. **Motion is an object that describes how a point moves along an axis, as time passes**. Shared motion allows this description to be shared across the Internet in real time, with high precision (milliseconds).

To clarify motion as a concept, consider the time-offset of a video element. As the video is requested to play, the value of the time-offset will increase in time. This is motion. To watch a video is essentially to move through the video file. Similarly, to pause the video is to stop moving. Crucially, motions are not limited to video, but rather pervasive in the media world. Slide-show presentation tools, frameworks for rich Web-based linear media, image galleries, time-series visualisation, video editing tools, broadcast TV, Web TV, maps and animations, they are all consumed by somehow **moving through the media**. Some of these offer controls allowing end-uses to direct their own motion. In the case of broadcast TV, end users may follow the motion through a public TV show, while the broadcaster maintains the controls. Motion is a generic concept, encapsulating matters of timing and control in linear media.

Motions are described in terms of *position*, *velocity*, *acceleration and time*. A motion may be started by *updating* the velocity to 2.0, and subsequent *queries* to the motion will confirm that the position increases by 2.0 each second. Crucially, motions may be shared across the Internet. This implies that **clients distributed across the Internet may query the state of a shared motion. If they do so at exactly the same moment (in real time), they will all agree (to a good approximation) about the state of the motion**.

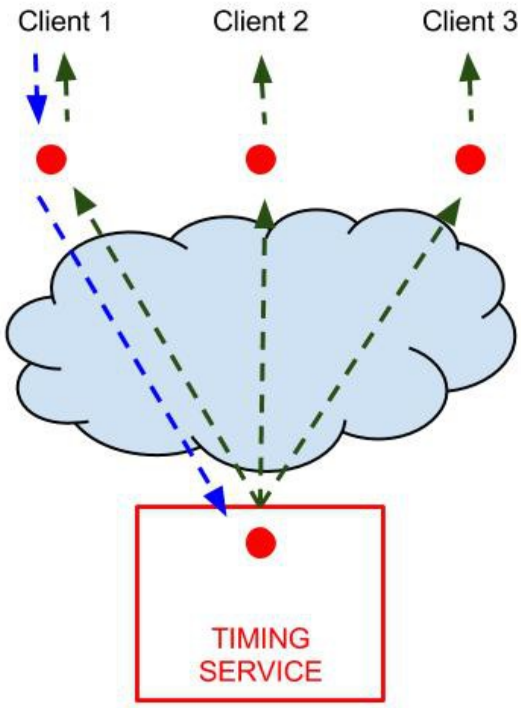

Illustration 7: Three clients synchronizing motions via an online service. Client1 requests the motion to change, and all clients are notified.

The shared motion implementation used in MediaScape is based on the technical concept of Media State Vectors (MSV). The blue arrows (downwards) illustrate a request to change the motion, which is hosted by the timing service. After a motion is changed, the timing service notify all connected clients, as illustrated with green arrows (upwards). Illustration 7 shows motion shared between three clients. High precision and scalability is achieved by leveraging a deterministic model for linear motion. Well known techniques for clock synchronisation are employed to mask effects of network latency and clock skew. However, this complexity is fully encapsulated, and as a consequence shared motion appears as a simple and intuitive concept for programmers.

Finally, shared motion matches nicely with the shared state protocol defined in chapter 6.

- Motions are represented as state on server (i.e., position, velocity, acceleration, time)
- Updates allow clients to change the motion.
- Motions will emit events when change notifications are received by the timing service, allowing swift reactions from application code.

9.3. Synchronisation of Continuous Media using Motion

Media players for continuous media such as audio and video need to keep track of progress, as well as allowing end users to pause and resume. In this sense, such media players already have motion, though usually internal and custom to the player. The central idea in motion-based synchronisation it that players should be directed by an external motion instead of an internal. Current players still have weak support for this. Still, as long as players support programmatic access to some kind of control primitives, this provides a rudimentary basis for synchronisation. In particular, HTML5 video and audio elements can be synchronised surprisingly well using external motion, considering that they are not at all optimised for this usage. With internal support for motion this would only be better. For example, given a “motion” attribute, video elements could be assigned an external motion. This is similar to how the “controller” attribute is used to assign external Media Controllers [MC]. Media Controllers implement rough synchronisation by coordinating play and pause signals for a group of HTML media elements. For synchronisation of multiple media elements within the same Web page shared motion offers more precise time-based synchronisation. Crucially, shared motions allow synchronisation of media elements

distributed across the Internet.

9.4. Synchronisation of Non-Continuous Media using Motion

Shared motion can also be used to control the presentation of non-continuous, time-referenced media, for instance subtitles. A simple technique is to check the motion periodically and then display relevant subtitles. A more sophisticated technique would be to calculate the time to the next event based on motion, and use a timeout to ensure that subtitles are introduced at exactly the right time. Using timeouts is also an energy efficient solution, as precise timing may require excessive, redundant polling, potentially draining the battery of mobile devices. In MediaScape we provide a **motion sequencer** as a generic mechanism for timing all sorts of non-continuous media. Given a set of points/intervals and a motion, the sequencer emits callbacks at the correct time, every time motion enters or leaves an interval, or passes a point. The sequencer encapsulates all the timing complexity and will always be correct according to motion. The sequencer also supports dynamic changes to the set of points and intervals, so that linear media may be safely edited without requiring a presentation reload. Finally, the sequencer is part of the solution for T4.1, ensuring that time-referenced data is made available at exactly the right time at the client side.

9.5. Task Outcome

The architecture of T4.3 was based on the idea that clients will be timing aware, using shared, application-specific clocks. Shared motion, a distributed timing-mechanism has been used for this purpose. Additionally, continuous as well as non-continuous media has been synchronised according to motions. This is easy for programmers, as the timing complexity is encapsulated by generic JavaScript libraries. This high-level architecture has been the basis for API specification and implementation in T4.3.

10. Conclusions

Work Package 4 has provided a solid and highly flexible foundation for online, multi-device Web application in MediaScape. The functional parts of WP4 has been exploited in a wide variety of technical prototypes, including the official MediaScape prototypes, thereby demonstrating wide applicability. WP4 has also created suitable and intuitive concepts and API's for developers, evidenced internally in the project by a low very level of issues and complaints reported back from developers. WP4 technologies have mainly just worked, except for the occasional bug that could be fixed easily and quickly. A commitment to the shared state model throughout the project has been an important factor behind this. Another factor has been an emphasis on solving the bigger problem by means of small, well-defined, self-contained and reusable building blocks.

A major outcome of WP4 is the initiative towards standardization of the Timing Object [TO], based on the shared motion approach and the timing tools used and verified within the MediaScape project. MediaScape has demonstrated that given the right concepts, multi-device Web applications may be built to exploit global timing in the millisecond domain, opening up the Web platform to a series of new applications and use cases, particularly relevant to online media and broadcasting. MediaScape has also demonstrated that these timing related capabilities can be made available to Web developers in a very simple and Web-friendly way, as flexible programming model allowing developers to work explicitly with timing and control, crafting application-specific timing logic with the help of generic and powerful programming tools.

11. References

[MSV]: *Ingar M. Arntzen, Njål T. Borch and Christopher P. Needham. 2013. "The Media State Vector: A unifying concept for multi-device media navigation". In Proceedings of the 5th Workshop on Mobile Video (MoVid '13). ACM, New York, NY, USA, 61-66.*

[TO] : Timing Object Draft Specification http://webtiming.github.io/timingobject/

[MC] MediaControllers http://www.w3.org/TR/html51/embedded-content-0.html#mediacontroller

[PRO] Promises http://www.w3.org/TR/2013/WD-dom-20131107/#promises

[AJAX] Asynchronous Web Applications http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/