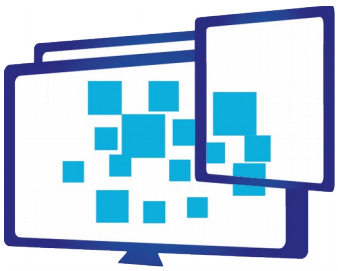


MediaScape
Dynamic Media Service Creation,
Adaptation and Publishing on Every Device

Call Identifier: FP7-ICT2013-10
Grant Agreement: 610404

WP4 – SYNCHRONIZE – Synchronization

Deliverable	D4.3
Title	Final implementation of multi-device synchronization services
Version	1.0
Date	19th November 2015
Status	FINAL

Restricted

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

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

Document Control

Version	Date	Author	Modifications
0.1	23 Jun. 2015	Mikel Zorrilla (VIC)	Template and structure
0.2	24.02.2015	Ingar Arntzen (NOR)	Initial Content
0.3	11.03.2015	Igor G. Olaizola (VIC)	Internal Review
0.4	14.03.2015	Ingar Arntzen (NOR)	Initial version finalized
0.5	18.10.2015	Njål Borch (NOR) Andreas Bosl (IRT)	Final version draft
0.6	17.11.2015	Chris Needham (BBC)	Internal Review
1.0	19.11.2015	Njål Borch (NOR) Ingar Arntzen (NOR)	Final version

Table of Contents

1. Executive Summary

Synchronization in multi-device applications has two main interpretations.

One interpretation is to do with consistency of shared data sources. For example, if an application depends on a shared playlist for media, modifications to this playlist must be propagated to all agents using it, preferably as quickly as possible. In WP4 we refer to such a playlist as ***shared data***. The concept of shared data thus implies **data synchronization**.

Another interpretation of synchronization is to do with temporal aspects. For instance, in a multi-device media application video, audio and subtitles may be split and presented collaboratively by agents on separate devices. In this case, synchronization relates to simultaneous playback of different media. In WP4 we refer to such timed operation across multiple devices as ***shared timing***. The concept of shared timing implies **temporal synchronization**.

In WP4 we explore both data synchronization and temporal synchronization. Though it would appear that these are entirely independent issues, a key contribution from WP4 is the combination of data and temporal synchronization techniques.

WP4 adopts the client-server architecture of the Web and adopts a **service-based approach** to shared data and shared timing. WP4 results are generic services dedicated to resource sharing in multi-device applications. As such, WP4 address challenges that are common to most multi-device applications. Furthermore, the focus on services implies that WP4 results can easily be added and integrated into existing applications.

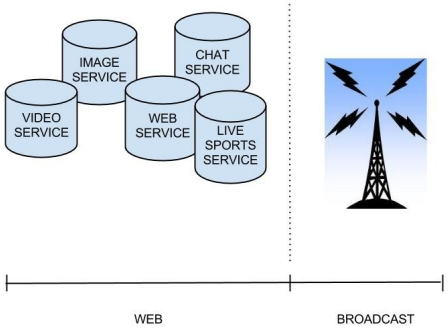
As well as core services, a more structured abstraction is needed for sharing of specific state commonly associated with an instance of a multi-device application, and its connected agents. In the project, we have defined the **shared context** of the application to include information about agent capabilities as well as running state for all agents of the shared context. This is realized through the **application context**, which is based on top of generic **shared data** services.

WP4 defines and implements services for shared application resources; including **application data** (T4.1), **application context** (T4.2) and **application timing** (T4.3).

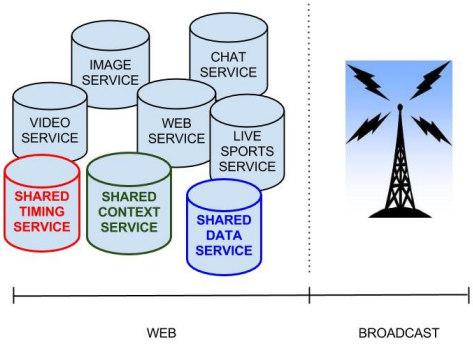
2. Introduction

2.1. Background

In modern broadcasting, traditional broadcast is often extended with parallel Web-based coverage. For instance, Web coverage of a major sports event might involve resources such as HTML, video clips, live sport results, user generated photos, and perhaps an interactive chat or a commenting service. The figure below illustrates how the broadcaster backend now is comprised of a number of independent servers or services, each specialized for a particular type of data and/or access pattern.



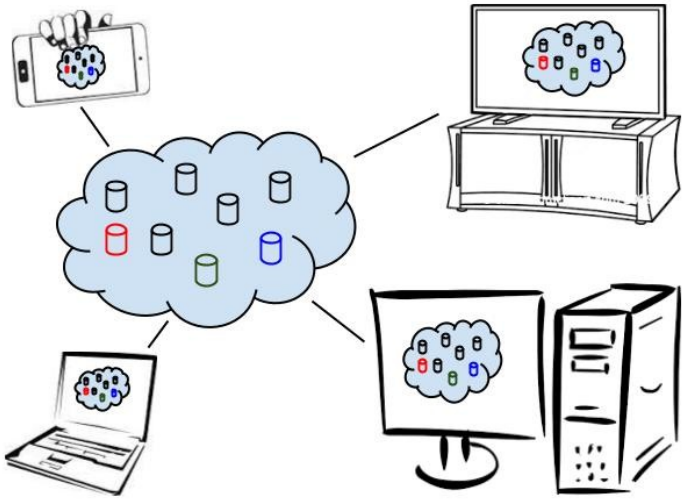
As specified in D4.1, WP4 aims to provide solutions for data synchronization and temporal synchronization within the classical client-server architecture of the Web. This ensures simple integration of WP4 contributions into current production systems of media providers and broadcasters. WP4 contributions provide specific types of services of special significance for multi-device Web applications. In particular, WP4 focuses on services supporting consistent resource sharing in multi-device applications, be it sharing of highly dynamic application data, application context or application specific timing and control. In WP4, this is labelled **shared data, shared context** and **shared timing**, as illustrated in the figure below.



Multi-device Applications : Model

The emphasis on a strict client-server architecture informs the model of multi-device application employed by WP4.

1. Multi-device applications are built from online resources identified by URLs. These resources are hosted by servers/services. Clients consume resources, monitor resources and interact with resources. Servers/services are assumed to be dependable.
2. Clients provide independent views into a running multi-device application. Devices may host different views into a single application, or they may all host the same view. UI-rendering is a client-side activity, essentially driven by time, user inputs and dynamic changes in server-side resources. Non-UI clients (robots or embedded devices) may also play part of applications by interacting with server-side shared resources.
3. Clients may connect to or disconnect from a running application at any time, without interfering with the application or with other clients (unless this is desired application behavior). Clients are stateless, so page reload is equivalent to disconnect-connect.
4. Servers do not maintain client state across sequential connections. This is in line with RESTful design principles.



The illustration above shows how four devices present individual views into a running multi-device application. Each device takes some subset of the available resources and use these to produce a presentation. If one of the resources is a shared playlist, modification to this playlist in one view will affect all views which depend on the same resource. Similarly, if the video playback position is a shared resource, pausing the presentation in one view causes it to be paused in all views.

Multi-device Applications : Definition

In WP4 resource sharing is understood to be central to the definition of multi-device applications.

From an end-user perspective, sensible multi-device applications likely make it easy for end-users to understand the relations between different devices, and the nature of their collaboration. Effective clues in this regard may be shared or related media content, shared sound or visual clues, causality across devices, or simultaneous operation. All of these aspects relate to consistent sharing of resources, be it data or temporal aspects.

Also, from a systems perspective, multi-device applications are defined with reference to resource sharing.

- there exists one specific online resource (identified by URL) that defines the application.
- any client connected to this resource is by definition part of the application
- to create a new application is to create a new application resource of this kind
- the application resource is one entry point for the application, and will (among other things) allow connecting clients to resolve which other resources and views are part of the application. There could be multiple entry points for one application.
- resources within an application may be public, global to the application and/or instantiated and accessed on a per-login basis.

2.2. Scenarios and Requirements

User scenarios for the MediaScape project, as well as the requirements derived from them, have influenced the design of 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

Req 4.1 Device Notification

The need for quick notification of state changes, as well as consistency in data synchronization motivates WP4 solutions for shared data.

Req 4.2 Timeline based synchronization

The need for synchronization of timed data according to a timeline (e.g. timed subtitles) drives

WP4 research in temporal synchronization. WP4 provides shared timing, allowing multiple devices to agree on time progression along a common timeline. WP4 also provides solutions allowing timed data (e.g., timed subtitles) to be correctly aligned with shared timing.

Req 4.3 Media Synchronization

The need for synchronization of audio and video across devices is also supported by shared timing mechanisms of WP4. In particular, WP4 explores client-side synchronization of HTML5 video relative to shared timing.

Req 4.4. Shared/Remote media control

The need for remote control of audio and video in MediaScape use-cases, as well as other time-sensitive aspects, indicate that shared timing is not only a read-only mechanism, but must support interaction and programmatic control. Use cases also highlight that the ability to remote control should not be limited to specific devices, but be available from different devices, simultaneously or at different times. This motivates a symmetric, server-based approach to remote controlling, where remote control is supported indirectly through interaction with shared server-side resources.

Req 4.5 Pause/resume/seek

The need to support common primitives in media navigation, such as pause, resume and seek, motivates the integration and support of such primitives with WP4 shared timing.

Furthermore, as WP4 shared timing is not limited to use-cases with audio and video, the mechanism aims to be expressive enough to be useful for other media frameworks as well, such animations, timed visualizations etc.

Req 4.6 Mash-up and Extensibility

The need to support use cases focusing on mash-up and extensibility has reinforced the commitment of WP4 towards the client-server model of the Web. In short, a hallmark of the Web is loose coupling, allowing a variety of resources to be mixed within a single page or presentation, without introducing any dependencies between those resources. WP4 architecture honors this approach. In particular, by modelling temporal aspects explicitly as shared resources, mash-up and extensibility applies not only to data, but also to temporal aspects.

Req 4.7 Resource resolving (new)

A need to quickly resolve relevant resources for different scopes has been identified during the last reporting period. For example, a group of users sharing experience require all devices of each one user to share state information for adaption (*user scope*), while they all might need a shared dataset as well, such as a playlist (*group scope*).

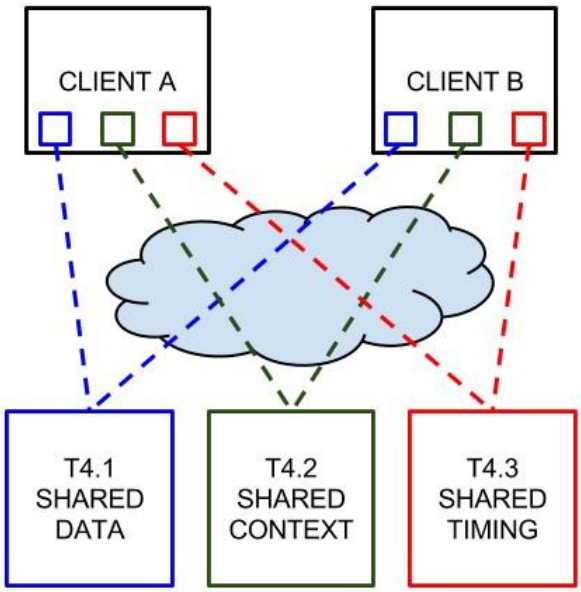
2.3. MediaScape Architecture

The WP4 architecture, as presented in D4.1 and summarized above (section 2.1 Background) is in line with the overall MediaScape architecture presented in D2.2 and D2.5. However, the WP4 architecture is limited with respect to MediaScape architecture, as it does not address solutions that make use of local network or other forms of network proximity.

WP4 provides generic services for shared data and shared timing, to be used by components developed in other WP's. WP5 explores generic services for multi-device adaptation. This work relies on WP4 for sharing of application context as the basis for adaptation. WP3 explores discovery, pairing and control both in a local network setting and for online scenarios. The online case might make use of WP4 shared data services. WP3 also contributes context information into the shared context.

2.4. Multi-device Synchronization Services Approach

Synchronization in multi-device applications relates to two distinct problems. One is synchronization of data for distributed consistency. The other is synchronization of timing in the interest of temporal consistency. WP4 addresses both these challenges. WP4 is divided into three tasks. Tasks T4.1 and T4.2 address data synchronization (application data and application context) whereas T4.3 is dedicated to temporal synchronization.



2.4.1. Task 4.1 Shared Data (IRT)

Shared data is a basic building block in MediaScape. Data synchronization is a central challenge in multi-device applications, especially when shared, server-side objects may be modified at any time. This task focuses on providing a fundamental building block for WP4 and MediaScape at large. A generic service is developed focusing on low update latency, quick onchange notification, consistency among observing clients as well as efficiently providing connecting clients with current state. The service is intended primarily for relatively small volumes of JSON data essential for the application. If application objects are large, the WP4 data sharing mechanism may still be useful in sharing references to such objects, meta-data or representations of change operations (i.e., active replication).

Generally, there are trade-offs between properties such as efficiency (latency/throughput), scalability and consistency. The shared data implementation follows a streaming-based or queue-based approach, where asynchronous update operations are essentially streamed sequentially through an online server. Notifications of changes are then streamed/multicast back to clients, where they may be used to maintain a local cache of server-side data, allowing all read operations to be resolved locally. Strict server-side serialization of update requests makes it easy to guarantee eventual consistency in local caches. Stream-based processing makes for a very efficient implementation. However, what you trade away is the classical synchronous request/response way of accessing server-side data. Instead, the programmer must be aware that update operations will not take effect locally until change notifications have been received from the server (delayed by network round-trip time). This asynchronous pattern for access to shared state is gaining popularity [RS, FSF, AS, TDIO] and is well aligned with modern GUI frameworks and the increasingly event based programming model of the Web.

Note also that the particular shared data implementation of WP4 is not intended as an optimal solution for every possible multi-device application. Instead, WP4 envisions a variety of shared state implementations with different properties with respect to

- data model (key-values, files, relational tables etc)
- data type (JSON, XML, string data, binary data, etc)
- scalability, consistency, efficiency, and reliability

Given a selection of shared data implementations, programmers would choose the one that best fits their problem, or employ different variants for different parts of application data.

The WP4 implementation of shared data is conceptually a simple service for sharing (key,value) pairs. As such, the service is of general value, and useful for different purposes and for different data types within MediaScape. In terms of research contribution, the final goal of this task is to demonstrate how to extend this basic service with multi-device time-sensitivity. This would improve consistency even more, by masking effects of network latency and ensuring that modifications are applied simultaneously by

different devices in a multi-device scenario. This requires integration with concepts developed in Shared Timing T4.3.

To support a dynamic creation of shared data resources a Mapping Service was introduced (Mapping Service API in the section below). The core functionality of the Mapping Service is to map from a **scope** to a resource. For MediaScape apps, both an application identifier (AppID) and a user identifier (UserID) are available. The scopes provided by the Mapping Service are:

- User scope: All agents of the UserID will share this resource regardless of which application is running.
- App scope: All agents of the given AppID will share this resource regardless of which user runs the application.
- UserApp scope: All agents of a particular user that runs the given application will share the resource. This is the chosen default mapping for WP5 activities, where application adaption is done for all active agents of a given user.
- Group scope: A mapping from a group identifier to a resource. Particularly useful for a shared session between multiple users.

The mapping service resolves resources matching the given scope, and creates the shared data resource if it is not already available.

The Shared Data and Mapping Service implementations are both integrated with an authentication service (Google OAuth is default, but can be configured easily to any other authentication provider).

Status

A stable implementation for Shared Data, backend (**mediascape/WP4/Server/**) and client-side library (**mediascape/WP4/API/mediascape/Sharedstate/**), according to the WP4 API specification (see section below) has been developed. The implementation is functionally complete and has been verified by test code and usage. The Shared Data API is well defined and has been stable the last 24 months. Repeated use and experimentation has not exposed design-level weaknesses.

2.4.2. Task 4.2 Shared Context (NOR)

This task focuses on providing a generic service for sharing of data related to highly dynamic application context. For instance, many multi-device applications might benefit from sharing sensor data or instrument data, such as accelerometer data or whether mobile screens are currently on or off. If available, such contextual data may be valuable input for a running multi-device application. In particular, dynamic context data is input to multi-device adaptation in WP5.

Sharing of contextual data is not very different from sharing data in general. For this reason shared context makes use of the generic service for shared data developed in T4.1. However, the shared context services also differs in some important respects.

- Shared context must bind contextual data to specific agents, so notions of agent identities and agent presence must be part of the abstraction.
- Sharing all possible contextual data, at all times, with all other agents, does not scale well. For this reason shared context explicitly supports definitions of interests through subscriptions accompanied by upstream (client-side) filtering.

Multi-device applications may come in many shapes and sizes. Still, we expect some commonality as well. In particular, most multi-device applications will depend/benefit from maintaining a representation of the current state of the application *itself*. We call this the *Context* of the application. The Context has two main parts, the *Agent Context* and the *App Context*.

Agent Context

The Agent Context provides direct access to information about the current agent (the one executing the software). It both allows information to be set, as well as being read. The Agent Context is also in charge of instruments, turning them on or off as needed, as well as doing conversions (downsample, derivative state etc). State is exported from the Agent Context to the shared Application context.

App Context

The App Context is shared between all agents of a user. In order to have this restriction, the App Context is based on the UserApp scope of the Shared State service.

The App Context provides presence information about agents, allowing all agents a full view of which agents a user is using. It also provides a simple way for the application to list capabilities of the different nodes, request updates to state changes and even set shared parameters. For example, agent A can discover that agent B supports “location”, and can request GPS updates by subscribing to the “location” parameter of agent B. This will ensure that the GPS on agent B is on (if allowed by the user), and the location of agent B will now be made available. Setting parameters is also allowed, making it possible to, for example, remote control the volume of the entire multi-device setup or the individual volume of each agent.

Shared Context Goals

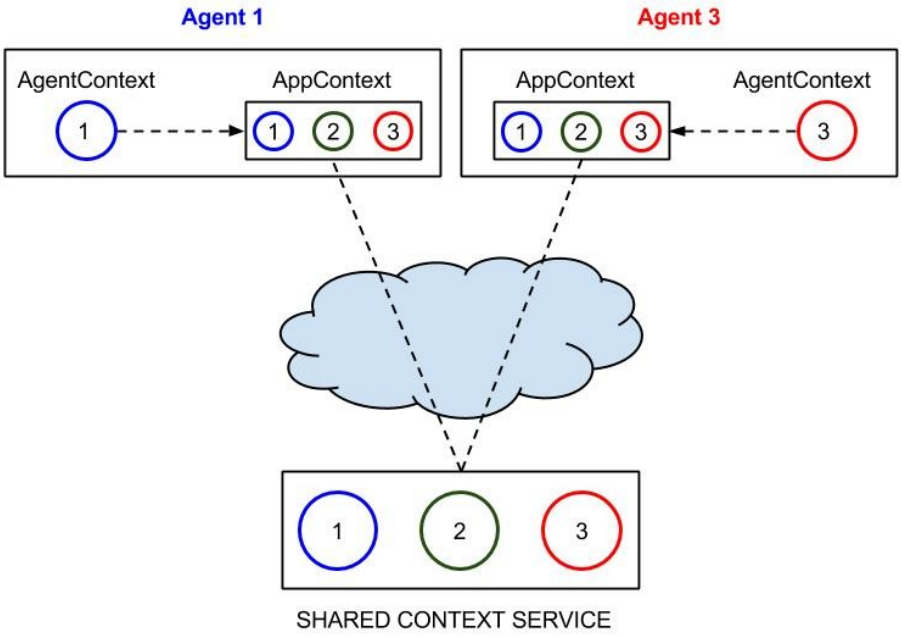
1. **Consistency/Agreement**. Much of the information included by the App Context and the Agent Context is highly dynamic. Still, it is essential that agents are provided a consistent view. Global agreement allows agents to make decisions safely, based on a truly shared context.
2. **Real-time.** When dynamic context information is changed, it is vital that this information is propagated quickly to all agents. Inevitably, the latency of the context sharing mechanism defines limitations for its applicability. For instance, consider agent A (PC) using accelerometer data from agent B (mobile phone) to control some aspect of media presentation.
3. **Fast Load and Persistence**. Context is lost whenever agents fail, or more trivially as web pages are reloaded. When agents connect and reconnect, the shared context must be loaded quickly so that the agent can resume its operation seamlessly. Context is persisted online so it may survive a temporary disconnect.
4. **Generic Mechanism**. Some types of contextual information may be relevant for some applications but irrelevant for others. The goal is NOT to provide a global definition of what information is included in shared context. Instead, we aim to provide a generic mechanism allowing application designers to include information relevant for the application. Application developer can thus subscribe to the parameters they find interesting while avoiding additional resource usage.
5. **Easy-to-use**. Discovery of agent capabilities automatically provides a wide range of contextual information, such as accelerometer, screen rotation, screen size. This makes it easy for application programmers to use advanced functionality in a very easy and intuitive way.
6. **Extensibility**. If a particular type of contextual information is not provided already, a mechanism is available for programmers to easily implement new capabilities and add it to the Shared Context.
7. **Lightweight**. In order to provide an efficient solution, the mechanism should limit collection and propagation of data. For this reason, the context mechanism includes the concept of subscriptions. Information is not collected or propagated unless at least one agent has subscribed to it. Subscriptions for context information may be turned on and off dynamically during application execution. Instruments will be turned on or off as necessary in order to limit battery usage and limit privacy implications.

Approach

The basic idea is to layer T4.2 Shared Context on top of T4.1 Shared State. This immediately ensures that goals 1, 2 and 3 are met for shared context, and serves as an independent validation of T4.1. In addition, it ensures that the concepts and APIs are kept consistent within the project.

As a Shared State object is used to hold the application context, the scope of the

SharedState must be suitable for an AppContext. For the MediaScape project, we determine that an AppContext is shared between all instances of an application for a particular user. This is due to the adaptation being based on which agents a user has active. Other scopes could of course be useful in other scenarios. In order to resolve the correct SharedState object, we have created a Mapping service, which can resolve a number of scopes. The UserApp scope is resolved and passed to the AppContext.



The figure above illustrates 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*.)

From a practical point of view, 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 resulting abstraction (layered on top of the shared data abstraction) is an

implementation of announce-subscribe-notify pattern, with persistence. Agents in this system announce *instruments* and may at any time publish new values for these instruments. These instruments may be subscribed to by other agents. Only if there is at least one subscriber will the value be propagated/shared. The persistence of the shared data backend ensures that a connecting agent will immediately be presented with a current snapshot of relevant context data (without having to query other clients).

Status
The AgentContext and AppContext has been implemented and integrated with the Mapping Service. Code contributed to **mediascape/WP4/API/mediascape/Sharedcontext/** and **mediascape/WP4/API/mediascape/Applicationcontext/**.

Integration of locally discovered nodes (WP3) into the AppContext object for a seamless integration for application programmers has not yet been completed. Such integration might make it easier for application programmers to use both local and global discovery. As none of the prototypes of the projects have this need, this solution has not been explored.

2.4.3. Task 4.3 Shared Timing

Introduction
As outlined in D4.1, the WP4 approach to shared timing is defined by the concept of shared motion. Shared motion is a service-based approach to distributed temporal control. Shared motions may be used as application-specific distributed clocks in multi-device applications, and as a mechanism for distributed media control. As such it is a perfect match with the server-based architecture of WP4, and solid foundation for addressing challenges of multi-device media synchronization and timed operation.

Given shared motion available in Web clients, the next step is to ensure that Web applications can easily build on this to support precisely timed behaviour -- directed by shared motion. In this regard, T4.3 focused on two central challenges. 1) **MediaSynchronization**: synchronization of continuous media (audio and video) directed by shared motion, and 2) **Sequencer**: synchronization of timed data (objects associated with time intervals) directed by shared motion. We contend that a wide variety of timing related challenges may be decomposed into a combination of 1) and 2).

Shared Motion Background
There are two main approaches to temporal synchronization; data-driven or clock-driven.

The data-driven approach is centered around concurrent data transfer. Essentially, by pushing data to multiple recipients at the same time, it follows that

execution/presentation will be roughly synchronized at the receiver end. TV broadcast is one example of this approach, where video signals are transported to a great number of clients over a broadcast network. Another example is web-coverage of live events, where small messages may be pushed real-time to connected web clients. A third example is pulse-driven protocols, where clock ticks or beats are broadcast to clients. In the music industry, the MIDI protocol has been used for temporal control of musical instruments for 30 years already, following this approach.

What data-driven approaches have in common is that timing accuracy is very sensitive to jitter and differences in network delay. This becomes especially problematic when synchronization is needed across very different communication channels. For instance, as broadcasters provide both traditional broadcast and web coverage of the same event, delay differences may be as much as 30 seconds between web content and broadcast.

Clock-driven approaches base their precision on system clocks (reliable) as opposed to network delay (unreliable). NTP (Network Time Protocol) has been around for a long time and provides a well-proven and widely deployed protocol for synchronization of system clocks. Unfortunately, synchronized system clocks is a not a reasonable assumption in the Web domain. In particular, few assumptions can be made for the system clocks of mobile devices. Furthermore, a read-only system clock is but a fundament for timing. A practical timing mechanism additionally must provide programming concepts allowing programmers to express application-dependent timing-sensitive behaviour.

Shared Motion introduces synchronized clocks into the Web domain, without making the assumption of synchronized system clocks. Instead, ad-hoc, application level clock synchronisation is performed every time a client connects to an online timing resource (i.e., motion). This guarantees availability of synchronized clocks for application logic on any connected device. The precision of such application-level clocks may be slightly coarser than expected for lower-level synchronization of system-clocks (i.e. millisecond as opposed to microsecond) Still, millisecond precision is suitable for the Web domain in general, and MediaScape use cases in particular.

In addition to clock synchronization, shared motion provides a programming model where applications may define and take direction from many different clocks (i.e., motions) and interact with them in order to change temporal aspects of the application. Efficient, low latency propagation of update requests and change notifications is also encapsulated by the same mechanism.

Shared Motion is an object that describes how an abstract *point* moves (in time) along a line. The motion describes in detail the position of the point, its velocity and acceleration at a given time. This information is represented by a vector [position, velocity, acceleration, time]. Shared motion is a simple concept encapsulating and solving four important technical challenges in multi-device applications;

1. Distributed media control (i.e. remote controlling)
2. Distributed simultaneity.
3. Distributed timing (i.e. controlling precisely when stuff happens)
4. Interoperability and Extensibility.

Please consult the design document D4.1 or the cited scientific publications for more detailed information on this concept.

Shared Motion in MediaScape

Shared Motion was invented and first developed as scientific prototype as part of EU project P2P-Next (FP7). Since then, a commercial company Motion Corporation has implemented a production quality service around this concept, and has agreed to provide MediaScape with free of charge usage (limited to research purposes) throughout the project period. As implementing yet another scientific prototype within MediaScape can no longer be considered a research advancement, WP4 instead focuses on further scientific exploration of this approach. T4.3 has done work to integrate and align APIs of shared motion so that they fit well with MediaScape conventions as well as provide an open source client library that is fully in line with the architecture and design of the MediaScape project. Additionally, as part of dissemination activities in WP7, MediaScape has created the Multi-device Timing Community Group [MTCG] within the W3C to suggest standardization of Shared Motion within the HTML standard [TO], and also advocate improvements to existing timing related issues.

Media Synchronization

A MediaSync object has been created based on a set of measurements of various browsers. It connects to an HTML media element and monitors its state. The state of the media element is then compared to the ideal state provided by the associated Shared Motion. Any discrepancy is compensated, either by using variable playback rates (if supported) or by skipping. As media elements are not designed for such external monitoring and control, browsers behave differently. The underlying operating system might also affect media elements, as do any hardware limitations.

The MediaSync object tries to automatically determine the best way to synchronize any stream on any browser. It has been tuned to provide relatively good results on as many browsers and devices as possible, which has led to a very simple solution. Better synchronization results can be achieved by tuning to specific browser/operating system combinations, but in the MediaScape project, ease of use and general usability has been regarded as more important. The MediaSync object also provides ample information about its current state and limitations, making it possible for the application programmers to detect problems and react to them (for example, suggest moving video to a better suited device).

Sequencer (previously Moving Cursor)

T4.3 has also focused on synchronization of timed data based on shared motion. This is important for implementation of time-sensitive storage abstraction in T4.1 as well as required by many use cases relating to timeline-based synchronization.

The term *sequencing* here refers to the process of translating a timed script into timed execution. The sequencer works on timed data, and is similar to the HTMLTrackElement. For example, given a set of timed subtitles, the essential function of the sequencer would be to provide the correct subtitle at the correct time, as well as remove it precisely when it's no longer valid. By emitting precisely timed "enter and "exit" events, the sequencer completely encapsulates timing complexity for programmers and consequently simplifies programming of linear presentations and other timed operations significantly.

The sequencer improves on the HTMLTrackElement in many ways. It is much more precise with respect to timed event upcalls, and it also operates consistently in the event of skipping on the timeline. In addition, it supports dynamic data during playback. This includes both addition and removal, as well as changes to the temporal aspects. In short, a subtitle may be asked to stretch in time during playback, while the subtitle is being presented, and the continued presentation will behave accordingly, without requiring any extra attention from the programmer. It also works on fully generic data, so no types or predefined restrictions exist.

Crucially, unlike the HTMLTrackElement, the sequencer is not tied to the progression of a HTMLMediaElement. Instead it integrates directly with shared motion. This immediately positions the sequencer as a highly valuable construct for dealing with any kind of timed data in multi-device applications, and it opens up for very flexible device adaptation as audio and video are no longer required as master track for a linear presentation.

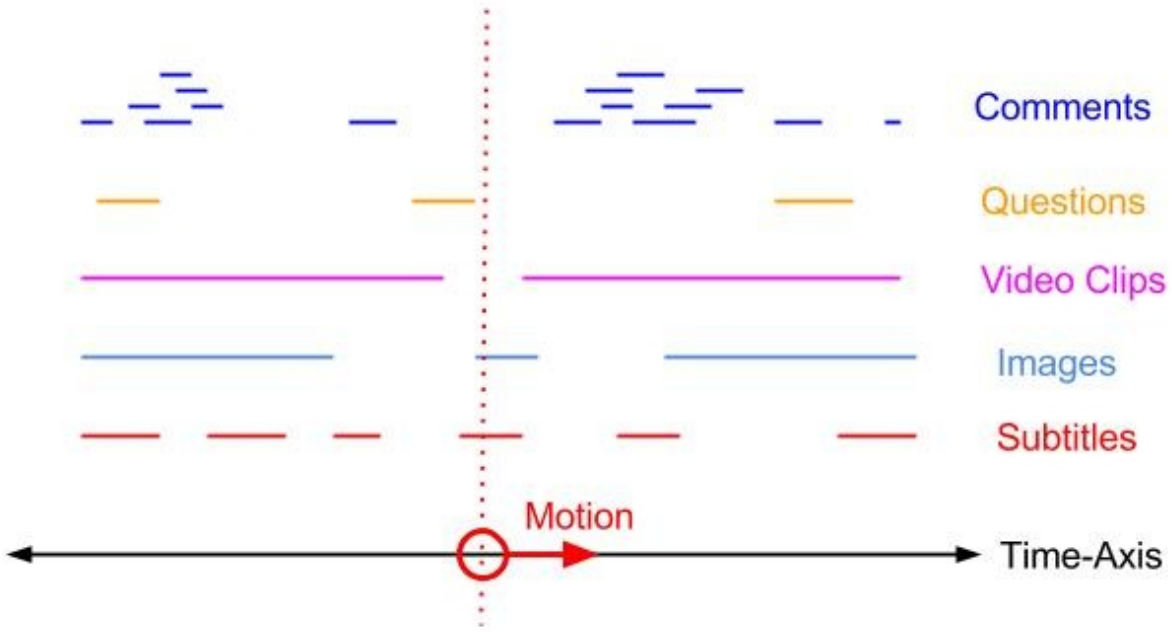
Background Sequencer

The sequencer works on *linear data*. By linear data we simply mean data that is somehow organised according to an axis, e.g., a *point* on the axis, or an *interval*. For example, a subtitle may be structured as follows, where properties *start* and *end* indicate validity related to the time-axis.

```
{
data : "Hello!",
start : 24.3,
end : 28.7
}
```

The figure below also illustrates linear data. It shows how a linear media presentation is defined by motion through linear data. The vertical line is the current position of the motion, defining at any time the "questions", "movie clips", "images", "subtitles" and "comments" that are *valid* at the current moment in time. Currently only a single

"subtitle" is valid. As motion continues forward, this subtitle will shortly cease to be valid, and need to be removed as motion enters a segment with no subtitle.



Music Box Analogy

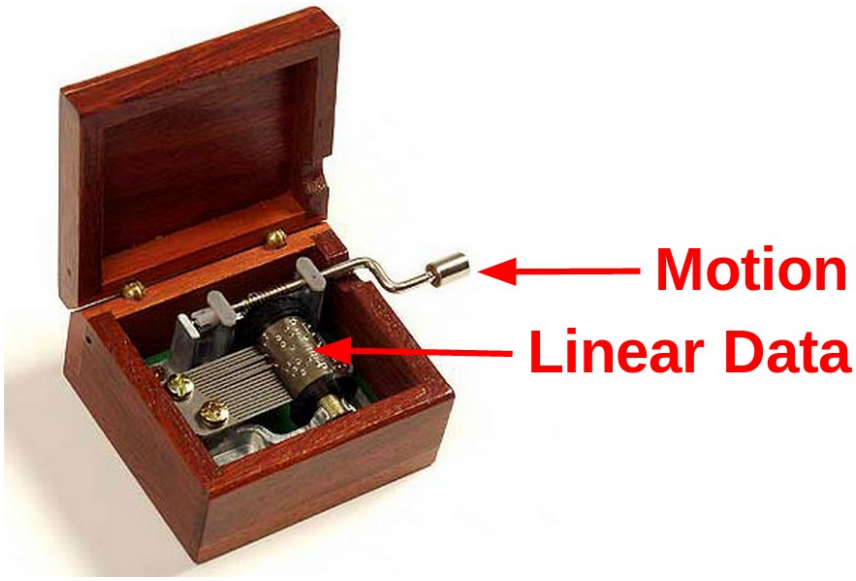
The music box below is another effective analogy for the sequencer. As the illustration shows, the music box takes two kinds of input, linear media and motion. If there is motion, then the music box outputs sounds at the correct moments in time to produce a melody.

The sequencer similarly works on linear data and motion, and it outputs callback invocations at the correct moments in time. Crucially though, the sequencer is superior to the music box in two important aspects:

First, the sequencer is designed to support linear dynamic data. This implies that the raw material of the linear presentation may be modified at any time during playback, in terms of both media content and timing aspects. This may for instance be done directly by viewers interacting with the presentation, or by a production team. The sequencer ensures that this can occur safely, and it fully encapsulates the complexity that arise from this.

Second, the sequencer runs on shared motion. It maintains time-consistency for any kind of movement supported by the shared motion, be it double speed, backwards, jumping ahead or even acceleration. Furthermore, by virtue of being based on shared motion, the sequencer may be remote controlled across the Internet. The sequencer may also play a

part in synchronized multi-device presentations, presenting the same linear data at multiple screens, or different (related) data sets in a multi-screen production.

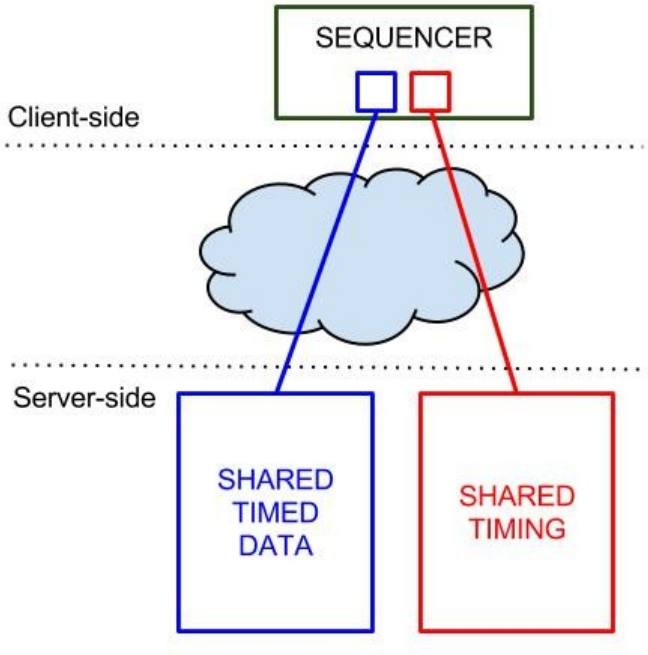


Definition

The sequencer encapsulates the concept of *temporal-validity* in linear data. A sequencer manages a collection of timed objects, or cues. A subset of these cues are said to be *active cues*, *valid* at the current moment in time. So, as motion progresses, cues will spontaneously appear and disappear from the collection of active cues, always at the correct moment in time (millisecond precision). These transitions are signalled by precisely timed of “enter” and “exit” events. The set of active cues are always maintained consistent with motion.

Architecture

In the context of MediaScape the sequencer makes use of two basic building blocks defined earlier in this document; shared timed/linear data and shared motion.



The figure above illustrates how the sequencer fits into the general architecture of WP4.

- The sequencer is a client-side construct, combining timed data and timing to produce correctly timed operation. In WP4 it is used with great effect to combine shared data resources with shared timing resources.
- Shared Timed Data (BLUE) and Shared Motion (RED) are managed separately and independently at the server-side (by specialised services), yet merged at the client-side by the sequencer to provide precisely synchronized behavior for timed data.
- The Sequencer is data-agnostic, but may easily be integrated with shared data in order to sequence timed data.

Programming Model

The sequencer, along with the underlying services for shared data and shared motion, provide the programmer with a novel, powerful, flexible, yet extremely simple programming model for multi-device linear or time-sensitive applications. The programmer is essentially left with two tasks:

- **Task 1 : Define Linear Data**
  - Define the data (JSON structure) and put it into shared data*. Map temporal aspects of timed data (e.g. “start” and “end” properties) to sequencer cues.
  - Update data (at any time) by overwriting it in shared state, or remove it.
- **Task 2 : Create a timing-sensitive viewer and connect it to the sequencer.**
  - The viewer will simply respond to events from the sequencer (“enter”, “exit”).
  - Implement DOM visualisation based on data format defined in Task 1.

* Note that the usage of shared data is not required for the concept of the sequencer to be useful. Any data source may be used.

In short, the programmer defines input data and output visualisation. Through the use of shared state, shared motion and the sequencer, the MediaScape platform effectively makes sure time-sensitive data can be applied (by the client) at exactly the correct time, by distributed agents. In addition, presentations may adapt dynamically to changes in data and motion. Also, the viewer (Task 2) does not need to be specifically aware of any timing complexity. All of that is encapsulated by the sequencer. Note also that only a basic understanding of the concepts, along with classical web programming skills, are required from the programmer.

This constitutes a generic, powerful, distributed and time-sensitive programming model. Consider that handlers implemented by programmers represent logic, and that the combination of shared motion, shared state and sequencers provide distributed execution of this logic, with the right data at the right time.

Status

The MediaScape API to shared motion is stable and WP4 use of the SharedMotion service provided by MotionCorporation is fully operational. The Sequencer and MediaSync libraries have well-defined APIs, and are fully implemented with Shared Motion. They are both in daily use, with few or no known issues. The standardization effort of Shared Motion is proceeding well under the name **Timing Object**. A draft specification [TO] is published at by the W3C Multi-device Timing Community Group [MTCG]. As part of dissemination and standardization work, MediaScape also have published open source JavaScript implementations of MediaSync [SYNC] and the Sequencer [SEQ].

2.5. Structure of the Deliverable

The tasks discussed in 2.4 will be presented each in their own section. Each section will document the API, feature code examples and references to code repositories. However, as shared data, shared context and shared timing are instances of a common pattern, the APIs have much in common. In WP4, this common pattern is labelled ***Shared State***.

This document first introduces an abstract Shared State API, and then details the APIs of each of the three tasks as special cases of this common pattern.

3.1 Abstract Shared State API
3.2 T4.1 Shared Data API
3.3 T4.2 Shared Context API
3.4 T4.3 Shared Timing API

3. Multi-device Synchronization Services

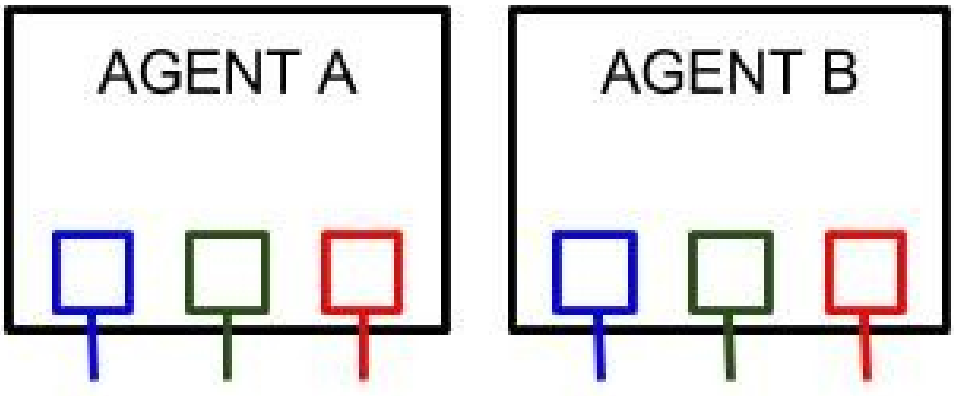
3.1. Abstract Shared State API

3.1.1. Goals and Purpose

Shared state solutions maintain distributed agreement concerning dynamic, online application resources in multi-device applications. Application resources may for instance include application data, context, data, or timing resources. Agreement about current state is maintained continuously, even if the resources are dynamically changing, e.g. updated frequently by connected clients. Please consult the design document D4.1 for further introduction to the Shared State programming model.

Shared state solutions have two parts; **1) an online service** and **2) a local proxy** for that service. The proxy provides a local, representation of the current state of the remote service, and programmers may always depend on this representation being updated as soon as possible.

Shared state solutions define a high-level programming model, where distributed agents/clients interact with the application and each other - not directly - but indirectly through a common data model. This includes **querying** and **updating** shared state, as well as reacting to **events** as shared state changes. This programming model hides complexities regarding connection management, message-passing, discovery and pairing. Furthermore, the shared state approach encourages clients to communicate directly with servers (reliable) instead of clients (unreliable).



The figure illustrates two agents with three shared state proxies each (blue, green, red). The proxies represent agents in the system, and are used to analyze agent capabilities as

well as interact with the state of the agent.

In this document we are not so much concerned with how shared state services are implemented, but focus on the API and the programming model that they expose. So, this document describes **the API that is provided by the local Shared State object at each device (proxy object for online Shared State resource)**. The API will mainly export methods for **querying** and **updating** shared state, as well as **events** to which event-listeners may be attached. Since MediaScape focuses on web-based multi-device applications, we describe a JavaScript API available in the MediaScape (MS) namespace. Implementations of Shared State are not required to have exactly the same API, but they will follow the same general pattern.

Initialisation

```
var ss = sharedState(url, token, options);
```

The proxy object is instantiated with a string url identifying a particular resource available on a shared state service. A token must be provided so that access privileges may be resolved by the shared state service.

ReadyState

Since the proxy object needs to establish a connection with an online server, the proxy will not immediately be ready to use. The property *readyState* will report the current status of the proxy at all times. In the event that shared state proxies are disconnected from the service, they will automatically try to reconnect, and the readyState will reflect these transitions.

```
ss.readyState;
ss.on("readystatechange", function (e) {});
```

Event listeners may be bound to the *readystatechange* event of the proxy. The readystatechange event is fired whenever readyState changes. It should be possible to register event handlers at any time - even before the proxy is connected.

In addition, an initial event is triggered by ss.on(“readystatechange”, handler), for the given handler only. This way the programmer can depend on the handler being called even if the handler was attached *after* the appropriate readyState (e.g. OPEN) was reached. This pattern is repeated for all shared state events (see further discussion below).

ReadyStates may be defined as follows.

```
var _states = Object.freeze({
   CONNECTING : "connecting",
   OPEN : "open",
   CLOSING : "closing",
   CLOSED : "closed",
});
ss.__defineGetter__("STATE", function (){return _states;});
```

We assume this struct to be associated with the **.STATE** property of SharedState instances.
The .STATE property should refuse assignment.

Here is some sample code:

```
var ss = SharedState();

// check if open
if (ss.readyState === ss.STATE["OPEN"]) {}

// print readystate
var onreadystatechange = function (state) {
   console.log(state);
   // or
   console.log(this.readyState);
}

// print possible states ()
for (var key in ss.STATE) {
  if (!ss.STATE.hasOwnProperty(key)) continue;
  console.log("key " + key + " value " + ss.STATE[key]);
}
```

ReadyState Implementation Guidelines

1. MediaScape does not define states globally. Instead different objects (sharedstate, sharedmotion etc) implement individual state definitions. The definition of states is available on the object using the STATE property.
2. In code do not refer to the values directly, instead reference by symbolic names. This way it is possible to change values or introduce new ones later on.
3. Use STRINGS - (not INTEGERS) as values. This way we explicitly DO NOT support expressions with > or < (essentially making use of the ordering). This is in line with common web practices. Using >, < expressions is not safe, because adding new states or changing values can have unintended side-effects if they affect ordering.

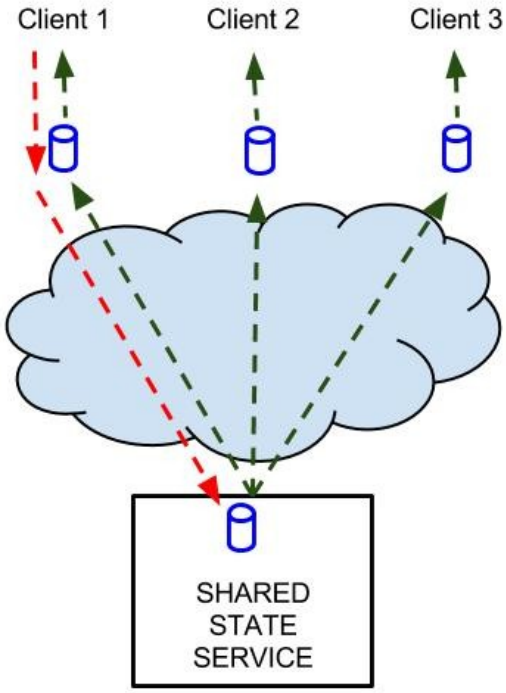
Update

```
var sent = ss.update(param, ... ,  options);
```

The update operation requests some kind of change to the shared state.For instance, items in a shared collection may be created, modified or removed. The update signature may be slightly different for different instances of shared state. Some APIs may also choose to expose more than one type of update operation, e.g. create and remove primitives.

The update operation simply sends a *request* to the shared state service.

The return value indicates whether the request was successfully sent or not. In particular, success does NOT indicate that the update has been performed. In contrast, an update operation does NOT take effect until it has been processed by the shared state service. The request may be lost before it reaches the service, or the service may even drop requests upon receipt. The effects of successful updates will be disseminated to all connected agents, including the agent that issued the update. So, after issuing an update request, an agent must expect to wait about 1 RTT before the effects become available locally.



The figure illustrates how an agent may issue update-requests (**RED**), and how the effects are made available to all agents equally, by broadcasting a change-event (**GREEN**).

No Update-Reply

The update operation does NOT include a handler for the asynchronous update-reply. In fact, there is no update-reply, only change-events. Programmers accustomed to Ajax-based web programming might find this surprising. They are used to request-reply interaction, the foundation of the Web. However, this is precisely the point of shared state. It is a higher-level abstraction hiding the complexities of lower-level request-reply interaction. In the shared state model there is only updates (actions) and events (reactions). In particular, change-events are handled the same way independent of which agent issued the update.

Batch Updates

```
var sent = ss.request().update(param, ... ,  options)
                       .update(param, ...., options)
                       .update(param, ..., options)
                       .send();
```

In a collaborative application it may be valuable to group multiple update operations into one, and have the shared state service process them all in one batch. This may avoid interleaving with updates from other clients. It would also ensure that unwanted intermediate states never occur.

Shared state services use the builder pattern to build a multi-update request, and then to send it. Shared state services must support such lists of operations in update requests. We recommend that shared state services always work on lists. This way the single update operation above is simply a special case, essentially a shortcut for ss.request().update().send();

Relative Updates

It is possible for shared state services to support update operations that are relative to current state at the server. For instance, the end result after an APPEND operation depends on what the state was before the operation. This is a relative update. Shared state services may support relative updates. For instance, options may be used to indicate if parameters are interpreted as absolute or relative.

Update Consistency

We expect shared state services to provide global ordering of all update operations. We also expect this ordering to be preserved through event notifications, so that all clients see the same sequence of state changes. Shared state solutions may offer more sophisticated consistency models, or such models may be built on top of a simpler one.

Query

```
var result = ss.query(param, ... ,  options);
```

Query is a ***local***, inexpensive operation. The local proxy process queries based on it's current state at any time. Repeated queries will reflect changes in the shared state. Application programmers may for instance choose to update the UI (DOM) based on event-listeners or periodic queries.

Event

Example

```
var handler  = function ( e ) {};
ss.on("change", handler, ctx);
ss.off("change", handler);
```

Shared state defines a change event, triggered whenever there is a change to shared state. Application programmers may register and unregister event-listeners using on/off primitives. Different instances of shared state solutions may export more event types. Optional parameter ctx is used as value for ***this*** in event callbacks. If not specified, ***this*** is the object from which the event was emitted.

Event handlers always expect a single event, corresponding to a single update operation. A batch update will result in multiple invocations of the handler, possibly one for each operation in the batch.

In shared state event handlers, the keyword **this** is always the source of the event, i.e. the shared state proxy.

Initial State from Event Handler

The following is a common pattern when attempting to update UI from shared state:

```
var current_state = ss.query();
update_ui(current_state);
ss.on("change", function(e) {
   update_ui(e.new_state);
});
```

- On load the current state is put into the UI.
- Next, subscribe to events so that future state changes will be fed to the UI.

This pattern may become a bit tedious as it is repeated for multiple subsets of the state,

and multiple UI elements. Fortunately, we can simplify this for the programmer. We require an initial event to be triggered within ss.on("change", handler), specifically for the given handler. This initial event produces the initial state (before any other events). This way, the above code snippet is reduced to:

```
ss.on("change", function(e) {
   update_ui(e.new_state);
});
```

Another implication of this trick is that the programmer does not have to worry about attaching the handler too late - i.e. "after the fact". If the programmer does this the handler will be invoked with the current state.

For this reason, we require shared state event handlers to deliver an initial event (only to the given handler) before any real events.

3.2.3.2 Shared Data API (T4.1)

3.2.1.Goals and Purpose

General purpose mechanism for efficient and consistent sharing of key,value pairs in multi-device applications.

3.2.2. Relation with MediaScape Architecture

The Shared Data service is a self-sufficient building block in the form of a backend service. It fulfills a simple and efficient, consistent sharing of key,value pairs for multiple agents. The introduction of agent presence was motivated by needs of the Shared Context module in T4.2.

3.2.3. API Definition

A shared dictionary is simply a collection of key-value pairs. The API is modelled after the LocalStorage API.

Shared Data Provider

A Shared Key-value store is hosted by a service provider. This provider is assumed to allow shared key-value stores to be defined and managed, for instance through a web interface. Each shared key-value store is identified by a URL. This provider management interface is not part WP4 API. Here we simply focus on how to interface with a shared key-value store, given a URL.

Initialisation

```
var ss = mediascape.sharedState(url, options);
```

Initialise with url and an optional options object

Options:

.reconnection	automatically reconnect, Default = true
.agentid	agentid used for this connection, Default = random
.userId	userId used for this connection, don't supply it if you are using a server based login
.getOnInit	sync the whole sharedState on start, Default = true

.autoPresence	sets presence to “online” on start, Default = true
.autoClean	removes context data from old clients, Default = false

readyState property and readystatechange event as described above.

Update

```
var sent = ss.setItem(key, value);
```

```
var sent = ss.removeItem(key);
```

```
var sent = ss.request().setItem(..).setItem(..).removeItem(..).send();
```

The function setItem will set a key to a value, removeItem removes the key from the dictionary. The dictionary allows multiple update operations to be grouped in one batch.

Query

```
var value = ss.getItem(key);
```

return value if key exists, else *undefined.*

```
var keys = ss.keys();
```

return list of all keys, may be an empty list

Event

Example

```
var handler  = function ( e ) {};
ss.on("change", handler, ctx);
ss.off("change", handler);
```

Event Types

“readystatechange”	whenever readyState is changed - e.g. disconnect
“change”	whenever a dictionary key is added or changed
“remove”	whenever a dictionary key is removed
“presence”	whenever a agent connects or disconnects

Change Event Properties “readystatechange”

e is simply the state value

Change Event Properties “change” and “remove”

.key	string
.value	value or null
.type	string, used in “change” events, “add” or “update”

Change Event Properties “presence”

.key	agentid as string
.value	value

Presence

If agentid is not supplied, a random one is generated. This will happen on every reload, so if you want to store it, put it in localstorage or remember it in some other way.

```
// Option for SharedState:
ss = mediascape.sharedState(url, {"agentid":"someid"});

// Set the state
ss.setPresence(state);

// React to presence changes
ss.on("presence", handler);
```

Example:

```
<html>
<head>
<script type="text/javascript" src="mediascape.js"></script>

<script type="text/javascript">

var sharedstate;

window.onload = function() {
  sharedstate = mediascape.sharedState("http://example.com/")
  .on("readystatechange", function() {
    console.log("READY");
```

```
    sharedstate.setItem("aKey","someValue");

  })
  .on("presence", function(e) {
    console.log("Presence: "  + e.key + " changed to " + e.value);
  })
  .on("change", function(e) {
    console.log(e.key + " = " + e.value);
  });
}
</script>
</head>
<body>
</body>
</html>
```

Mapping Service

The mapping service provides URLs for Shared State channels, and will create these if they are not already present. To provide a flexible solution, applications can request different scopes for such channels in order to simplify both implementation of the services and ensure a flexible and scalable solution.

Scopes

Based on login (need userid and appid)

- Scope *User*
  - All agents of the current user regardless of application
  - Example: All private agents: device state (off, idle, in user by app X)
- Scope *UserApp*
  - All agents of the current user for the given application
  - Example: Shared Context
- Scope *App*
  - All users of the given application
  - Example: Data points for all users of an application, e.g. comments

Group service:

- Scope *Group*
  - All participants of a given group
  - Example: Collaborative viewing, Quiz

Initialisation

```
var map = mediascape.mappingService(mappingServiceUrl, options);
```

Initialise with *mappingServiceUrl* and an optional options object

Options:

.maxTimeout	maximum time to wait for a response from the server in ms, Default = 2000
.userId	userId used for this connection, don't supply it if you are using a server based login

Event

Like for Shared Data, with only "readystatechange" as an event type.

Request a User mapping

```
var mappings = map.getUserMapping(appId, scopeList);
```

returns a Promise including the requested scopes or all available scopes {scope1:"url", scope2: "url", ...}

appId - id of the current App
scopeList - a list of the requested scopes ["user", "userApp", "app"].
If a url for a scope doesn't exist it will be created.

Request a Group mapping

returns a Promise with {groupId:"url"}

groupId - id of the group

Example:

```
<html>
<head>
<script type="text/javascript" src="mediascape.js"></script>

<script type="text/javascript">

var mapping, userAppSS;
var APP_ID = "TheUniqueIDofThisApplication";

window.onload = function() {
  mapping = mediascape.mappingService("http://example.com/")
```

```
  .on("readystatechange", function() {
    mapping.getUserMapping(APP_ID, ["userApp"]).then(function (data) {
      userAppSS = mediascape.sharedState(data.userApp);
      ...
  });
}
</script>
</head>
<body>
</body>
</html>
```

3.2.4. Implementation

Shared data uses websockets for efficient two-way communication between clients and servers. Update notifications are transferred from client to server, with update notifications in the opposite direction. Update notifications are broadcast to all connected clients. Connecting clients will immediately get a the current state so that local caches can be initialized correctly. State queries are always resolved locally, as all clients always have all data.

Keys are strings. Values are JSON.

Code available in GitHub
mediascape/WP4/API/mediascape/Sharedstate
mediascape/WP4/API/mediascape/Mappingservice

3.3. Shared Context (T4.2)

3.3.1. Goals and Purpose

General purpose mechanism for efficient and consistent sharing of context data in multi-device applications.

3.3.2. Relation with MediaScape Architecture

Extension of shared data from T4.1.

3.3.3. Definition

This documents the main steps in using the shared state mechanism; how capabilities are advertised, how capabilities can be subscribed to, and how events are propagated and received. Again, we consider an example of sharing accelerometer samples within a multi-device application.

Initialising AppContext and AgentContext

```
var appContext = mediascape.applicationContext(url, options);
```

AppContext is initialised by providing a URL to the shared state server (see T4.1). The constructor creates a singleton object. Options is a dictionary. If no {“agentid”:<string>} is given, a random ID is used (will be generated every time).
References to appContext and agentContext may be obtained as follows:

```
var appContext = mediascape.appContext;
var agentContext = mediascape.agentContext(); // agent's own agent context (private)
var agents = appContext.getAgents(); // all agents in AppContext (including self)
var selfContext = agents.self; // agent's own agent context (shared)
```

Agents is a map of agentContext objects, indexed by agentID.

Context Providers

App Context and Agent Context is populated by *Context Providers*. Context providers are identified by a name, e.g. ‘videoplayer’ or ‘accelerometer’. The purpose of provider is simply to provide a namespace and logical separation between independent parts of the context.

- Context providers implement and advertise their capabilities
- Context providers can be queried for status information
- Context providers encapsulate complexity with subscriptions and events
- A set of ready-to-use Context providers is provided for common sources of context

information
- New Context providers may be implemented and loaded

Loading Context providers and Advertising Capabilities

Context providers provide capabilities to an agent. As such, they will add capabilities to the agentContext when they are loaded. For that they use the setCapability function.

```
agentContext.setCapability(capability, value);
```

Add or set a capability for this agent. E.g. setCapability("audio", "supported"), setCapability("location", ["fine", "coarse"]).

The pre-programmed context providers are loaded automatically by the agentContext if they are supported on the system. If the programmer wants to add custom context providers, a load function has been made available.

```
agentContext.load(component);
```

A Context provider is a JavaScript object implementing a specific interface. We describe this interface below in the section "Implementing Custom Context providers". The programmer should not have to know this interface in order to load a predefined Context provider.

MediaScape provides a number of existing Context providers for sensors, browser information, battery information, sensors etc. Some additional Context providers have been made as well, such as "shaking" to notify when a device is being shaken, and some Context providers resample or perform additional calculations, such as downsampling accelerometer data and calculating rotation values for a phone that is held up rather than laying on a table.

Discovering Capabilities

The programmer may need to figure out which capabilities are provided by the agent.

```
agentContext.capabilities();
```

Returns a list of all capabilities provided by this agent. (Union of all capabilities provided by all Context providers of this agent.)

NOTE: Capabilities are shared by default, so any agent can query its AppContext (locally) to discover remote providers of particular capabilities.

Subscribing to Capabilities

If an agent wants to interact with a capability, it needs to subscribe to that capability. For instance, below we subscribe to the capability “geolocation”.

```
agentContext.on("geolocation", func, options);
agentContext.off("geolocation", func);
```

The on() and off() functions register event handlers for the given context parameter. Options is a dictionary (associative array) or null (depending on particulars for the component in question). For instance, options for geolocation could be something like {“quality”:”precise”} or similar.

Subscribing to capabilities is a simple local operation. However, much happens behind the scenes. The subscription is shared and persisted via shared context. As the agent that provides this capability (subscribee) is notified of the new subscription, it registers the subscription locally. If this is the first subscription for the component, underlying sensors may have to be started. Similarly, as the last subscriber unsubscribes, the sensor will be shut down.

NOTE: This approach is also known as *upstream* subscriptions. Upstream subscriptions avoid wasting bandwidth for events that no-one subscribes to. A second benefit of this approach is that power-consumption is kept at a minimum as sensors do not have to be started until demand is explicit.

NOTE: Since subscriptions are persisted through shared state, the subscription may survive a short period after an agent disconnects. This implies that subscription will survive a reload or a short outage. However, if an agent has been disconnected for a while, the subscription will eventually be invalidated.

Interacting with Capabilities.

Once a subscription has been registered for a capability, the subscribing agent can interact with that capability locally. The capability essentially becomes shared state, so the subscriber can receive events (through the provided event handler), or simply query the state of the capability.

```
agentContext.setItem(key, value);
agentContext.keys(); // list of defined keys
```

```
agentContext.on(key, function(key, val) {...});
agentContext.off(key, func);
```

The functions are:

setItem	Set a key to a value
keys()	List the available keys
on(key, handler)	Subscribe to updates of a given key
off(key, handler)	Unsubscribe to updates of a given key

Dynamic App Context

App Context must adapt dynamically as agents connect and disconnect. Agents may also change dynamically by advertising new capabilities, or by revoking capabilities. The “agentchange” event on App Context provide such notifications.

```
appContext.on("agentchange", func(e) {e.agentid; e.agentContext));
```

The agentchange event is generated when an agent appears, or when the metadata about an agent changes (e.g. added a capability). If agentContext is set to null, the agent is offline.

Global Variables

The shared context mechanism is designed to support representation of fairly advanced instruments and sensors. However, sometimes all that is needed is just a global variable to be part of shared context. If so, there is no need to create a context provider for that. Instead, by virtue of being layered on top of shared state, shared context supports the same API for setting and getting global variables.

```
appContext.setItem(key, value);
var keys = appContext.keys(); // list of defined keys
appContext.on(key, function(key, val) {...});
appContext.off(key, func);
```

Set a shared parameter for the entire application. Value can be a string or an object. This is basically a global variable, writeable for all and is not connected to any particular agent. Example use could be “mode” set to “testing”, “demo”, “production”, “mute”:false etc. The API is exactly the same as for the agentContext described above.

Implementing Custom Context providers

The following example shows how to implement a simple Context provider that provides a "counter" capability. It essentially provides a single value that changes over time. When registered, the Context provider allows other agents to monitor this value.

```
var counterTimer;
var provider = {
  "counter": {
    init: function() {
      // this is set to the agentContext.
      this.setCapability("counter", "supported");
    },
    on:function() {
      var pos = 0;
      // this is set to the agentContext.
      // Called when the "instrument" should be turned on, start the counter
      counterTimer = setInterval(function() {
        pos++;
        ac.setItem("counter", pos);
      }, 1000);
    },
    off: function() {
      // this is set to the agentContext.
      // Called when the instrument should be turned off
      clearInterval(counterTimer);
    },
   val: -1 // optional initial value
   }
}
agentContext.load(provider);
```

Load a Context provider. The "on" function will be called when someone subscribes to "counter", "off" is called when all agents have unsubscribed. The counter in the example always starts on 0 and will count the seconds until all agents have unsubscribed. As updates are propagated automatically by the applicationContext (as a result of the setItem call), all subscribing agents will have their registered callback function triggered once per second.

Turning Context providers On and Off

When a capability receives its first subscriber, the underlying context provider is automatically turned on by calling the "on" function. Similarly, when the last subscriber disappears, the context provider is turned off.

Additionally, the Context provider can specify a requestHandler(key, agentID) that will be called when the Context provider is to be turned on. This allows the Context provider to prepare the user for requests made by the browser (e.g. "Allow access to your location?"), do a simple access control or analyse instruments more closely.

Example

```
var myContext = agentContext();
var appContext = appContext(url, ...);

appContext.self.on("volume", function(val) {video.volume = val});
appContext.on("mute", function (val) { video.mute = val});

var agents = appContext.getAgents();
var numAgents = agents.length();
...
```

This small example shows an application where video can be muted via a global variable. In contrast, volume levels are shared per agent, so each agent has its own volume setting. Agents can monitor and control each other's volume settings. The number of participating agents is available at any time simply by counting the number of agents.

3.3.4. Implementation

JavaScript library wrapping of shared key-value store.

Code available in GitHub
mediascape/WP4/API/mediascape/Agentcontext
mediascape/WP4/API/mediascape/Applicationcontext

3.4. Shared Timing (T4.3)

3.4.1. Goals and Purpose

Explore general purpose mechanism for efficient and consistent sharing of timing resources (i.e., motions) in multi-device applications.

3.4.2. Relation with MediaScape Architecture

Backend Service. Basic self-sufficient building block.

3.4.3. Definition Motion API

A single motion has a unique URL and implements the shared state programming model.

Shared Motion Provider

Motions are provided by a service provider. This provider is assumed to allow motions to be defined and managed, for instance through a web interface. Each motion is identifiable by a URL. The provider management interface for motion is not part of the WP4 API. Here we simply focus on how to interface with shared motion, given a URL.

Initialisation

```
var motions = MS.get_msvs(url_list, token);
```

The parameters are:

url_list	List of URL's to Shared Motions.
token	Access control token

As the MediaScape project uses Shared Motions from the Motion Corporation as opposed to implementing a separate version, the motions are instantiated remotely and used like this:

```
var app = MCorp.app("appid");
app.run = function() {
  // Use app.motions here
}
window.onload = app.init;
```

The Motion Corporation provide a web page (http://dev.mcorp.no) where application developers can create applications. This will generate the appid in the example above, and the developer can define Shared Motions for the application. Both private (unique for each user) and public (shared between all users) motions can be defined. A name is given to each motion (e.g. "private", "shared", "volume", "video" etc). In the instantiated app

object, app.motions contains a mapping between these names and the actual Shared Motion objects. For example, app.motions.private will be instantiated according to the "private" motion as defined on the mcorp developer page.

As Shared Motions are provided by the Motion Corporation for the project, the get_msvs() function described above is not used directly within the project, and other motion providers will likely package motion instantiation similarly.

Update

```
var sent = motion.update(p, v, a);
```

parameters p,v,a are position, velocity and acceleration that you want to apply to the motion. If any of these parameters are null or undefined, this aspect of the motion will not be changed. Batching updates on a single motion makes no sense, as only the last update will take effect.

Query

```
var result = motion.query();

var position = result.pos;
var velocity = result.vel;
var acceleration = result.acc;
```

Request the current state of the motion. Repeatedly querying the motion will reveal that the position changes in time, if velocity is non-zero.

```
var isMoving = motion.isMoving();
```

Shorthand method to check the direction of the motion. Returns -1 for backwards, 1 for forwards and 0 if not moving.

```
var info = motion.getState();
```

Info is an object with the following properties:

.range	[start, end] list elements are floats	motion only within range
.skew	float seconds	clock skew relative to motion server
.latency	float seconds	latency to motion server
.src	string	URL of msv

.readyState	int	readystate
.connType	string	“ws” for WebSocket, “http” for HTTP
.cred	string	“r” or “rw”
.lastVector	[pos,vel,acc,ts]	the initial vector that was before
.vector	[pos,vel,acc,ts]	the initial vector of the current motion

Event

Example

```
var handler  = function (e) {};
motion.on("change", handler);
motion.off("timeupdate", handler);
```

Event Types

“readystatechange”	whenever readystate changes - e.g. disconnect
“change”	whenever the msv is changed
“timeupdate”	fires periodically while there is motion (approx. 5hz)

Change Event Properties (identical to query result)

.pos	float	Position
.vel	float	Velocity
.acc	float	Acceleration
.ts	float	Timestamp (client clock)

The Shared Motion API is different from the draft specification of the Timing Object as published by the W3C Multi-device timing community group [TO], but are functionally equal. We have chosen to keep the current implementation while the specification matures.

3.4.4. API Definition Media Sync

Initialisation

```
var sync = mediascape.mediaSync(htmlElement, motion, options);
```

The htmlElement is a HTMLMediaElement (i.e., an audio or video tag), motion is a Shared Motion and a number of options can be given:

Option	Default	Description
skew	0.0	How many seconds should be added to the motion before synchronization. Calculate by start point of element - start point of motion.
automute	true	Automatically mute the element if playback speeds are outside of comfortable levels.
mode	“auto”	Force “skip” to avoid trying variable playback rate. “vbpr” forces use of variable playback rate regardless of success. “auto” tries to detect variable playback rate support, defers to skipping if unable to synchronize smoothly.
debug	false	If true, log to console, if a function, the function will be called with debug info.
target	0.025	The target synchronization point in seconds. For variable playback rate this is in effect 0.0. With a lower number, media sync will synchronize tighter to the motion, but the user experience can be hampered by incessant skipping. A higher value will make sync appear easier to reach, but will more likely be off. The default is frame accuracy or lip-sync.
remember	true	Remember the last experience on this device. It improves subsequent playbacks in particular on devices that does not have working variable playback rate support. If the debug option is set, remember is automatically set to false and any stored values removed. Recommended left on true for production systems, false for development and testing.
loop	undefined	Loop the content at the given time. E.g. set to “30.0” to loop the content every 30 seconds.
duration	mediaelement.duration	When looping, at which point do we loop. Different browsers might report different mediaelement.duration, so specifying duration is likely necessary.

Use

```
sync.setSkew(skew);
var skew = sync.getSkew();
```

Update the skew of the media or read the current skew value.

```
sync.setOption(option, value);
```

Set an option on the go. Typically this could be to enable/disable debugging on the go.

```
var method = sync.getMethod();
```

Returns “vpbr” for Variable Playback Rate or “skip” for skipping mode.

```
sync.setMotion(sharedmotion);
```

Change the motion used for media element control. This can be used for example to easily switch between a private or a group motion.

Events

```
var handler  = function ( e ) {};
sync.on(event, handler);
sync.off(event, remove_handler);
```

The MediaSync object emits a number of events which can be subscribed to or unsubscribed from using the on() and off() functions.

Event	Description
skip	Called whenever the element is made to skip
mode_change	Called when the element changes mode. On subscribe, the current mode is always provided as an initial event.
muted	Called when muting is toggled. Requires automute option to be true. Initial event is given with the current state.
target_change	The MediaSync object can change it’s target dynamically to respond to thrashing or if the device is consistently unable to meet the target.
sync	Triggered when in or out of sync

Example

```
var sync = mediascape.mediaSync(document.getElementById('player'), motion);
```

3.4.5. API Definition Sequencer

A public API documentation for the Sequencer is maintained by [MTCG] at [SEQ].

The motion sequencer works on a collection of **cues.** A **cue** is essentially a (**key**, **Interval**) pair, where **key** is a unique (string). The sequencer outputs **enter** and **exit** events at the correct time, according to **motion** and the **Interval**. Events include **cue** information. The sequencer API and function is similar to the HTMLTrackElement. Application programmers may use unique keys from an application-specific data model.

Intervals

Intervals are expressed by two floating point values [low, high]. Infinity or -Infinity may be used as interval endpoints. If low and high are equal, the Interval is said to be a *singular* **point**. Intervals may or may not include its endpoints; [a,b], [a,b>, <b,a], <a,b>. This is defined by optional boolean flags **lowInclude** and **highInclude.** Intervals are immutable objects. Modification of a (key,Interval) association simply requires is to be replaced with a new.

Events

The main purpose of the sequencer is to trigger the events every time motion (i.e. position) **enters** or **exits** an Interval. (This is also true if the Interval is a singular point). By encapsulating this timing complexity in a generic mechanism, the programmer is left with the much simpler task of defining Intervals and reacting to sequencer events.

The sequencer guarantees that events are always emitted at precisely the correct time (error O(1ms)). This guarantee holds even as Intervals are dynamically added, modified or removed during playback. Sequencer events will include information fully describing the occurrence (see below).

If multiple Interval endpoints are bound to the same point, events will be emitted according to the following precedence, given that direction of motion is forwards. If direction of motion is backwards, the precedence order is reversed. With respect to precedence, we distinguish Intervals that represent *singular* **points** (low === high) from *regular* **intervals** (low < high).

1. **>** exit **interval** with open exit-point
2. ***[*** enter **interval** with closed enter-point
3. ***[*** enter **point**
4. ***]*** exit **point**
5. ***]*** exit **interval** with closed exit-point
6. **<** enter **intervals** with open enter-point

Active Cues

The sequencer also continuously maintains a list of **active Cues**. An Interval is active if interval.low <= motion.position <= interval.high. In other words, if the motion is *inside* the Interval, that Interval is said to be active. For instance, a subtitle may be active from 42.4 sec to 49.7 sec. More generally, the union of active Intervals defines the active state for any point in a linear presentation. So, cues with intervals are great for expressing state in linear presentations.

Sequencer does not have its own readyState. Consider it always ready, or consult the readyStates of the underlying motion.

Sequencer Module

```
var Sequencer = mediascape.Sequencer.Sequencer;
var Interval = mediascape.Sequencer.Interval;
// inheritance function -- see integration with data
var inherit = mediascape.Sequencer.inherit;
```

Initialisation

```
var s = new Sequencer(motion);
```

Interval

```
var i = new Interval(low, high, lowInclude, highInclude);
```

low and *high* parameters can not be given in the wrong order. *lowInclude* and *highInclude* are optional, but also sensitive to ordering. These define whether the low and high values are included in the Interval or not. For instance, the following variations may be constructed [low,high], [low, high>, <low, high] and <low,high>. The default is to include low, but not high: [low, high>

Cue

A cue defines an association between a string key and an Interval object. The Sequencer returns cue objects as follows.

```
var cue = {
  key: "key", // string key
```

```
  interval : object // Interval object
};
```

AddCue / RemoveCue

```
s.addCue("key", new Interval(12.4, 14.2));
```

addCue either creates a new cue or replaces an old cue. In the latter case the sequencer takes care of appropriate cleanup after the old cue. So, if the programmer wants to change an existing cue, for instance by changing interval.*high*, simply overwrite the cue with the old value for interval.*low*, and a new value for interval.*high*.

If addCue causes changes to the collection of active spans, a sequencer event is triggered with the appropriate exit or enter event.

```
s.removeCue("key");
```

removeCue simply removes a cue. If the cue was active, an exit event is emitted as the cue is removed from the set of active cues.

```
var r = s.request()
  .addCue("key1", new Interval(23.56, 27.8))
  .addCue("key2", new Interval(27.8, Infinity))
  .removeCue("key3")
  .submit();
```

The sequencer allows multiple cue operations to be grouped in one batch.

ActiveCues

```
s.getActiveKeys(); // returns list of keys
s.getActiveCues(); // returns list of cues [{key: “key”, interval: object}]
s.isActive(key);  // returns true if key is currently in the set of active cues else false
```

At any moment in time, the set of active cues are defined by the position of the motion at that time. The Sequencer maintains a set of active cues internally. This will change dynamically in response to motion. Emitted “enter” and “exit” events are always consistent with set of active cues.

Search

The sequencer additionally supports efficient search over its collection of cues.

GetCuesByPoint

```
var cueList = s.getCuesByPoint(searchPoint);
```

searchPoint is a float. The function returns a list of all cues, where the cue interval covers the given search point.

GetCuesByInterval

```
var cueList = s.getCuesByInterval(searchInterval); // returns list of cues
```

searchInterval is an Interval object. The function returns a list of all cues, where cue interval overlaps with the given search interval.

GetCuesCoveredByInterval

```
var cueList = s.getCuesCoveredByInterval(searchInterval); // returns list of cues
```

searchInterval is an Interval object. The function returns a list of all cues, where cue interval is covered by given search Interval.

Events

Events callbacks provide event arguments as an **eArg** object.

```
var handler  = function (eArg) {};
eArg = {
  key : string, // unique string key
  interval : object, // Interval object
  data : object, // used only for sequencer specialization
  point : float, // point entering/exiting cue interval
  pointType : string, // "low"|"high"|"inside"|"outside"|singular
  dueTs : float, // ideal time for event to be emitted
  delay: float, // delay relative to ideal time
  directionType : string, // "backwards"|"forwards"|"nodirection"
  verbType: string, // "enter"|"exit"
};
```

The Sequencer provides five distinct event types, “enter”, “exit”, “events” , “change”, “changes”.

“enter”

This event is emitted every time a cue enters the collection of active cues. This may be triggered dynamically by motion (i.e. playback or skipping) or by modifications made to the cues through .addCue(). The event concerns a single cue.

“exit”

This event is emitted every time a cue exits the collection of active cues. This may be triggered dynamically by motion (i.e. playback or skipping) or by modifications made to the cues through .addCue() and removeCue(). The event concerns a single cue.

“events”

```
var handler = function ([eArg]) {};
```

This event emits a batch of “enter” and “exit” events. Note that the parameter passed to the handler is a list of eArg objects. A batch of eArg objects may be produced if multiple cue intervals are associated with the same endpoint, or if if addCue/removeCue operations are processed in a batch. This event allows the programmer to take the entire batch of events into consideration when implementing application specific reactions. Use eArg.verbType to distinguish between “enter” and “exit” events.

“changes”

This event is emitted when some modification affects an active cue, without causing it to become inactive (i.e. “exit” event not triggered). The change event is helpful for visualization of the set of active cues, ensuring that visualized objects can be created, modified and removed through the same eventing mechanism.

“changes”

Batch of “change” events, similar to “events”.

Subscribing to and un-subscribing from Events

```
s.on(eventType, handler, ctx);
```

on()

Registers a handler for a specified event type.

- eventType : string : {“enter”, “exit”, “events”, “change”, “changes”}
- handler : function object : event handler
- ctx : object : specify context for *this* object during handler invocation.

Additional remarks

- The sequencer emits both enter and exit events even for singular point intervals.
- “enter” and “events” convey the state of the sequencer and will provide initial state by emitting an initial event after s.on() has been called. This ensures that application code can always depend exclusively on the series of enter/exit events to provide the correct state.
- “delay” is currentTs - dueTs. Note that delay in not a direct measure of the effectiveness of the Sequencer. This is because dueTs is defined by the timestamp of the motion. Whenever the motion is updated, it will be delivered a little too late for the clients. This delay shows up in the delay property.

Sequencer Data Integration

The sequencer is data agnostic and should be useful independent of the particular data model an application uses, as long as the data can be represented as unique keys associated with Intervals (or singular points).

The Sequencer can be used directly, but also supports integration with a specific data model. Such specialization is done through inheritance. Two methods may be implemented by subclasses to allow this.

```
s.loadData () {};
s.getData(key) {};
```

loadData()

This method is called by the sequencer constructor and allows the programmer to load cues from the application specific data model and register them with the sequencer.

getData(key)

This method is used by the sequencer whenever it needs to resolve the mapping from key to object in data model. The sequencer uses this so that it can make objects from the data model available in enter/exit events - under the e.data property.

Example : ArraySequencer

This example makes an array sequencer for arrays with timed data.

```
/* Timed data */
var array = [
  { data: 'A', start: 0, end: 1 },
  { data: 'B', start: 2, end: 3 },
  { data: 'C', start: 4, end: 5 },
  { data: 'D', start: 6, end: 7 },
  { data: 'E', start: 8, end: 9 },
  { data: 'F', start: 10, end: 11 },
  { data: 'G', start: 12, end: 13 },
  { data: 'H', start: 14, end: 15 },
  { data: 'I', start: 16, end: 17 },
  { data: 'J', start: 18, end: 19 },
  { data: 'K', start: 20, end: 21 },
  { data: 'L', start: 22, end: 23 },
```

```
  { data: 'M', start: 24, end: 25 },
  { data: 'N', start: 26, end: 27 },
  { data: 'O', start: 28, end: 29 },
  { data: 'P', start: 30, end: 31 },
  { data: 'Q', start: 32, end: 33 },
  { data: 'R', start: 34, end: 35 },
  { data: 'S', start: 36, end: 37 },
  { data: 'T', start: 38, end: 39 },
  { data: 'U', start: 40, end: 41 },
  { data: 'V', start: 42, end: 43 },
  { data: 'W', start: 44, end: 45 },
  { data: 'X', start: 46, end: 47 },
  { data: 'Y', start: 48, end: 49 },
  { data: 'Z', start: 50, end: 51 }
];


var ArraySequencer = function (motion, array) {
  this._array = array;
  Sequencer.call(this, motion);
};
inherit(ArraySequencer, Sequencer);

ArraySequencer.prototype.loadData = function () {
  if (this._array.length === 0) return;
  var r = this.request();
  for (var i=0; i<this._array.length; i++) {
    r.addCue(i.toString(), new Interval(this._array[i].start, this._array[i].end));
  }
  r.submit();
};


ArraySequencer.prototype.getData = function (key) {
  return this._array[parseInt(key)];
};
```

The inherit function is supplied by the sequencer module (see below). ArraySequencer is also available in the sequencer module.

Sequencer UI Integration

The sequencer may control UI if enter/exit events are implemented with appropriate effects in the DOM. The following example shows how a viewer may be tied to a DOM element, allowing active cues to be written and removed to/from the DOM at the correct time.

```
var viewer = function (sequencer, elem) {
  var key = undefined;
  var enter = function (e) {
    key = e.key;
    elem.innerHTML = JSON.stringify(e.data);
    console.log(e.toString());
  };
  var exit = function (e) {
    if (e.key === key) {
      elem.innerHTML = "";
      key = undefined;
    }
  console.log(e.toString());
  };
  sequencer.on("enter", enter);
  sequencer.on("change", enter);
  sequencer.on("exit", exit);
};
// Create ArraySequencer and connect to UI
viewer(new ArraySequencer(motion, array), document.getElementById("viewerid"));
```

Integration with Shared State

A special SharedStateSequencer is provided for integration with ShareState. It is assumed that “start” and “end” properties represent timing information

```
var SharedStateSequencer = function (motion, state) {
  this._state = state;
  Sequencer.call(this, motion);
};
inherit(SharedStateSequencer, Sequencer);

SharedStateSequencer.prototype.loadData = function () {
  var self = this;
  this._state.on("change", this.onChange, this);
  this._state.on("remove", this.onRemove, this);
};

SharedStateSequencer.prototype.getData = function (key) {
  return this._state.getItem(key);
};

SharedStateSequencer.prototype.onChange = function (e) {
  if (e.value.hasOwnProperty("start") || e.value.hasOwnProperty("end")) {
    var op = this.addCue(e.key, new Interval(e.value.start, e.value.end), e.value);
  }
};

SharedStateSequencer.prototype.onRemove = function (e) {
  this.removeCue(e.key, e.value);
};
```

SharedStateSequencer is also available in the sequencer module. The SharedStateSequencer is also an example of how to integrate with dynamic data sources.

Note that in “onChange” and “onRemove” - the data values from the events are supplied as the data property to the addCue operation. For addCue this is strictly not necessary, as the sequencer may get the data by calling getData(key). However, in the special case where you have a batch operation with multiple addCue calls for the same key, intermediate values will not be visible to this.getData(). If those values are provided by the event (e.value) you can pass them on to addCue to make them become available in sequencer events.

Similarly, in “onRemove” e.value is passed as the second parameter to removeCue. This way the removed

data value (which is no longer available from getData(key)) can be included in sequencer events. This is optional.

Implementation

Shared motion uses websockets for efficient two-way communication between clients and servers. Update notifications are transferred from client to server, with update notifications in the opposite direction. Update notifications are broadcast to all connected clients. Connecting clients will immediately get the current state of motions so that local caches can be initialized correctly. Queries to motions are always resolved locally, as all clients always have all data.

The Shared Motion client-side library provided by Motion Corporation, and adapted for MediaScape by NOR. The server-side implementation is hosted by Motion Corporation.

MediaSync is JavaScript library wrapping Shared Motion and HTML5MediaElement. Provided by Motion Corporation.

Sequencer is JavaScript library wrapping Shared Motion provided by NOR.

Code is available in GitHub
mediascape/WP4/API/Sharedmotion
mediascape/WP4/API/MediaSync
mediascape/WP4/API/Sequencer

MediaScape namespace location

mediascape.Sharedmotion
mediascape.Mediasync
mediascape.Sequencer

4. Conclusions

This documents the final implementation of MediaScape synchronization services.

Task 4.1 (Shared Data) documents the design, API and implementation of an essential building block for multi-device applications, a key-value service optimised for efficient sharing of highly dynamic data. This concept has already proven its value in different circumstances, providing an effective and shared mental model for the MediaScape project. Furthermore, both the design and implementation have proven to be easily extensible with additional features as new demands has emerged.

Task 4.2 (Shared Context) documents design, API and implementation of a generic service for sharing highly dynamic, agent specific contextual information in multi-device applications. The implementation has successfully been realized using services provided by task 4.1, and has been adopted by other work packages in the project.

Task 4.3 (Shared Timing) adopts an approach to temporal synchronization and control based on Shared Motion, a generic multi-device timing mechanism for the Web. Temporal synchronization in Tasks 4.3 relies on a timing service provided by Motion Corporation, and an open source client side library has been created according to MediaScape conventions and requirements. Access to Shared Motion has allowed Task 4.3 to advance quickly, focusing its efforts on temporal synchronization of timed data. The Sequencer is the first contributions of this task. This concept encapsulates all complexity associated with timed presentation of linear data in multi-device applications, without adding any complexity for the programmer. The concept even supports modifications to linear data during playback. The MediaSync library is the second contribution. MediaSync allows multi-device playback with millisecond precision, in regular HTMLMediaElements.

The provided systems are relatively mature and already represent a significant scientific advancement. Finally, WP4 is already devoting much attention to dissemination of research results, in particular W3C standardization. In particular, a draft specification has been developed by MediaScape and [MTCG], proposing the timing object [TO] as fundamental concept for Web-based multi-device timing, control and temporal interoperability in the Web. The design of the timing object is inspired by shared motions, and the timing object also supports integration with shared motion through the concept of timing providers [TO]. MediaScape WP4 has contributed Sequencer and MediaSync libraries to W3C Multi-device Timing Community Group [MTCG] to support standardization work associated with the timing object.

5. References

[RS] “Reactjs” http://facebook.github.io/react/
[FSF] “Full-stack Flux”, Pete Hunt, https://www.youtube.com/watch?v=KtmjkCuV-EU
[AS] “Apache Samza”, http://samza.apache.org/
[TDIO] “Turning the Database Inside Out”, Martin Kleppmann, https://www.youtube.com/watch?v=fU9hR3kiOK0
[TO] “W3C Draft Specification Timing Object” http://webtiming.github.io/timingobject/
[MTCG] “W3C Multi-device Timing Community Group” https://www.w3.org/community/webtiming/
[SEQ] “Sequencer”, https://webtiming.github.io/sequencer/, https://github.com/mediascape/sequencer
[SYNC] “Media Sync”, https://webtiming.github.io/mediasync/, https://github.com/mediascape/mediasync