Proceedings of the ACM on Human-Computer Interaction (PACMHCI), 2017 (to appear)
https://dl.acm.org/pub.cfm?id=J1598

INFEX: A Unifying Framework for Cross-Device Information Exploration and Exchange

REINOUT ROELS, ARNO DE WITTE, and BEAT SIGNER, Vrije Universiteit Brussel, Belgium

In our daily lives we are witnessing a proliferation of digital devices including tablets, smartphones, digital cameras or wearable appliances. A major effort has been made to enable these devices to exchange information in intelligent spaces and collaborative settings. However, the arising technical challenges often manifest themselves to end users as limitations, inconsistencies or added complexity. A wide range of existing and emerging devices cannot be used with existing solutions for cross-device information exchange due to restrictions in terms of the supported communication protocols, hardware or media types. We present INFEX, a general and extensible framework for cross-device information exploration and exchange. While existing solutions often support a restricted set of devices and networking protocols, our unifying and extensible INFEX framework enables information exchange and exploration across arbitrary devices and also supports devices that cannot run custom software or do not offer their own I/O modalities. The plug-in based INFEX architecture allows developers to provide custom but consistent user interfaces for information exchange and exploration across a heterogeneous set of devices.

CCS Concepts: • **Software and its engineering → Development frameworks and environments**;

Additional Key Words and Phrases: Information transfer, information exploration, cross-device interaction

ACM Reference Format:
Reinout Roels, Arno De Witte, and Beat Signer. 2017. INFEX: A Unifying Framework for Cross-Device Information Exploration and Exchange. *Proc. ACM Hum.-Comput. Interact.* 1, 1, Article 1 (January 2017), 26 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Over the last decade we have seen an increase in the number of electronic devices that we use in our daily lives. While devices such as smartphones, tablets, digital cameras or laptops are already widespread [1], we witness the rise of new types of electronic devices including smartwatches, fitness trackers and other wearable technology. As stated by Rädle et al., "*We are witnessing an explosive growth of the number and density of powerful mobile devices around us. However, their great majority are still blind to the presence of other devices and performing tasks among them is usually tedious*" [24]. For example, if we want to transfer music from a smartphone to an audio-enabled sports watch for a workout, we often have to use a computer to retrieve the music from the phone and manually copy the songs to the watch by using vendor-specific software such as iTunes which complicates the process.

Some devices might have built-in WiFi to share content via email, cloud services or social networking. However, this approach has its own shortcomings as it requires additional user

2573-0142/2017/1-ART1 $15.00
https://doi.org/0000001.0000001

interactions and results in unnecessary delays given that a user is piggybacking another medium for quick data transfer. More importantly, this method only works if the receiving end also supports the chosen transportation method. In practice, this enables information exchange between modern smartphones, tablets and computers but excludes a large amount of other commonly used devices such as older devices, devices without a display or devices that do not run third-party applications. Nevertheless, just because a device does not support the previously mentioned approach does not mean that it can only be accessed when it is wired to a computer. For instance, even decade old cellphones allow access to resources such as contacts, pictures or music via the wireless Bluetooth protocol. Similarly, sports-related wearables such as heart rate monitors often use Bluetooth for providing access to the gathered data and recent digital cameras might come with built-in WiFi for accessing captured pictures. However, incompatible hardware characteristics, data formats and communication protocols limit developers and researchers to use only specific devices and interaction methods are usually tailored to cope with technical restrictions rather than focusing on the user experience. The HCI community has identified that low-level technical complexity often results in serious end user issues [7] and we see an opportunity for improving information exchange across a heterogeneous set of devices.

We present the INFEX framework for cross-device information exploration and exchange which offers extensibility and reusability on the device detection and connectivity as well as on the end-user interaction level. INFEX implements common communication protocols and access methods as plug-ins and acts as a mediator to handle device detection and information exchange. This allows application developers to use a higher-level interface for information exchange where every device, protocol or media type can be used in the same way, encouraging a uniform and consistent user experience. In contrast to existing approaches, the presented framework is highly extensible, designed to prevent some of the infrastructure issues mentioned by the HCI community and also supports devices without a display or I/O modalities. The functionally offered by INFEX allows both researchers and commercial developers to build applications for exchanging information between previously incompatible devices with a minimal amount of effort, thereby allowing them to focus on the user interface and potentially novel interaction techniques.

We start by describing different solutions for cross-device information exploration and exchange and discuss some of their limitations in the background section. We then define a number of goals and requirements for cross-device information exchange before introducing the architecture of our INFEX solution. After providing some details about the implementation of INFEX, we present a specific use case which unifies different solutions that have been presented in the background section and enables information exploration and exchange across previously incompatible devices, protocols and media types. A critical discussion of the presented INFEX framework and an outline of potential future work is followed by some general conclusions.

2 BACKGROUND

A unifying solution for information exchange between different devices involves many aspects, both on the technical as well as on the usability level. Note that we are not the first ones to notice the inefficiency of user-driven data exchange between different devices [21, 24]. In this section we discuss the body of existing work and identify a number of limitations of existing solutions which are going to be addressed by our approach.

2.1 Device Discovery

In order to exchange information between devices, the devices need to be aware of each other's existence. From a technical perspective this is trivial for cases where devices are connected via a cable. However, the transfer of information becomes more challenging when we aim for more user-friendly

wireless approaches. The detection of participating devices can, for example, be achieved in a peer-to-peer manner as seen in the RELATE [10] interaction model where custom ultrasound dongles [13] are used for relative positioning. In most existing approaches, the device detection is performed by some external infrastructure and not by the participating devices themselves. HuddleLamp [24] proposes a purely optical solution that uses RGB and depth data from an RGB-D camera to detect devices that have been placed on a surface. Other vision-based solutions include the tagging of devices with fiducial markers for camera-based tracking [18] or the use of a device's front-facing camera to detect fiducial markers on the ceiling [19, 29] to inform the underlying platform about a device's presence and position. Also wireless technology such as Bluetooth is used for detecting the presence of devices as seen in Connichiwa [30] and BlueTable [36] or RFID and NFC are applied in [27, 31, 32].

2.2 Connectivity and Integration

Another issue investigated by related work is the fact that the technical complexity of integrating a device into a network-based system often negatively affects the user experience. Different devices use different incompatible technologies and protocols. Further, technical details such as an IP address or port number are required to establish a connection. In order to address this issue, Frosini et al. [8] introduced the concept of *connectors*. When a device registers, its configuration contains information on how the framework can communicate with the device. The configuration describes which connector to use by specifying a connector type together with any relevant details such as the type of connection (e.g. HTTP) as well as an IP address and port number. Devices add themselves to a session by scanning a QR code but frameworks such as Conductor [12] extend this idea by using additional sensors such as NFC tags or accelerometers for detecting devices that are bumped together. Although some initial setup is required, these approaches make it easy to add the device to a session. Some of the previously mentioned solutions using RFID or NFC store the device's Bluetooth MAC address on the tag in order that the necessary information for a Bluetooth connection can be retrieved when a tag is detected [31, 32].

JCAF [3] is a infrastructure framework for context-aware applications and allows different sensors to be integrated in a network for context processing. However, the framework relies on Java RMI for retrieving data and only supports Java-enabled devices. DynaMo [2] is a framework for handling dynamic multimodality. In order to deal with a variety of input hardware, the framework uses Cilia [9] for mediating events captured from input devices before mapping them to specific actions. The mediation layer allows the framework to retrieve data via different protocols but after mediation the events can be processed in a protocol and format independent manner.

2.3 End-User Interactions

Ideally, interactions in a cross-device setting should be intuitive and uniform. From a usability perspective it is clear that the spatial component plays an important role in developing intuitive data exchange systems. Most academic work requires users to get the involved devices close to each other or even make them touch each other for exchanging information. As summarised by Marquardt et al. [21], these types of actions have been coined as *proxemic interactions* in the context of ubiquitous computing and are based on the study of people's understanding of spatial relationships [11, 22]. Note that there are implications on the technical level since a device's position, boundaries and orientation needs to be tracked for interactions that make use of spatial information.

Rekimoto's Pick-and-Drop [25] uses digital pens to allow users to pick up a digital object (e.g. a file) on one device's display and drop it to another device's display to copy or move the object. Note that this type of interaction reduces the complexity of data transfer over a network and offers an intuitive interaction similar to moving a physical object. Rekimoto later proposed a solution

that uses projection to display associated digital information around objects placed on a table surface [26]. The use of everyday physical objects for information sharing or as an interface to digital information sources has been investigated in Embodied Data Objects [33] or DroPicks [14]. The limited range of RFID and NFC technology further enables proxemic interaction. For instance, Seewoonauth et al. [31] and Sánchez et al. [27] discuss interaction techniques that use RFID tags for transferring files between different devices. A user can, for example, select an image on the phone and transfer it to a computer over Bluetooth by reading an RFID tag attached to the computer [32].

Dippon et al. [6] suggested the use of an interactive surface for exchanging photos on smartphones. Users can select an image from their local gallery for display on the interactive surface and then use the surface to drag it on the graphical depiction of another phone in order to transfer the image to the corresponding physical device. Dachselt and Buchholz [5] investigated the use of gestures to "throw" content or user interfaces from mobile phones to nearby displays. In the opposite direction, Schmidt et al. [28] describe a cross-device interaction style allowing mobile phones to pick up digital content from a surface by physically touching the content with the phone. Other systems that use large surfaces in combination with smaller devices for information exchange and collaboration include Activityspace [16], Dynamo [17], Impromptu [4] and WeSpace [35]. SurfacePhone [37] eliminates the need of an interactive surface by integrating projection and touch detection in the phone itself and turning it into a mobile workspace that can interact with other nearby workspaces. There has also been some work where non-standard devices have been used for information exchange. For example, WatchConnect [15] explores interactions (including content transfer) between a custom-built smartwatch and an interactive surface.

2.4 Limitations of Existing Solutions

The presented related work proposes various improvements for transferring content between devices and covers technical contributions as well as new interaction techniques. Nevertheless, a number of interesting observations can be made. Our findings in terms of information exchange are related to infrastructure shortcomings that have been documented by the HCI community. We briefly introduce these shortcomings using the terminology defined by Edwards et al. [7] and show how they manifest themselves in the previously discussed related work.

2.4.1 Constrained Possibilities. Due to technical complexity or limitations and the fact that dedicated software has to run on a supported device, almost all of the presented use cases for user-initiated information exchange limit themselves to laptops, smartphones and tablets only. Unfortunately, this excludes a large range of existing and emerging devices that cannot run custom software (e.g. music players, fitness trackers or other wearable technology). Additionally, existing frameworks and systems often assume that participating devices have a display and offer some input modalities such as a touch or keyboard-based input, which again excludes various popular devices. Similar issues can be identified on the protocol level where communication is usually restricted to either WiFi-based socket connections or Bluetooth. Finally, although a lot of the presented scenarios share these technical characteristics and have overlapping technical requirements, they are normally not extensible or reusable and often limited to a specific use case or media type (e.g. the exchange of photos is a common use case). Another example is the UPnP protocol which represents a more generic protocol for information exchange but still excludes many older or emerging devices from participation [7].

2.4.2 Unmediated Interaction. The technical complexity at lower levels often manifests itself as limitations, inconsistencies or added complexity to end users. For instance, in order to integrate a device in the network a user might have to set up the device by installing software and entering the IP address of a central server. The more protocols and devices are combined, the higher the risk

that low-level details affect the user experience. The user interface or interaction method might further change depending on the device or content type since for technical reasons they are treated differently .

2.4.3 Interjected Abstractions. Abstractions made on the technical level can also propagate to higher levels even if they do not contribute to the interaction model presented to the end user. For example, an end user might have to find and install drivers or adapters to add support for a specific device or content type, when it could have been as easy as enabling a checkbox in list. In a more literal sense, when considering developers as end users of a framework the interface provided by the framework might rely on abstractions that require specific technologies or programming languages. Some frameworks might require developers to interact via plain old Java objects (POJOs) or Java RMI which offers benefits for Java-based clients but excludes clients without Java support.

The shortcomings identified from the related work show that there is plenty of room for improving cross-device information exploration and exchange. For example, in the presented related work a lot of the end-user interaction has been implemented from scratch due to the lack of extensibility and reusability in existing solutions. As a result, researchers and developers often do not spend time to address the technical complexity but rather only support the most common devices. Further, most existing solutions are limited to a fixed set of devices and interactions and it is difficult integrate and unify existing approaches for exchanging information across devices supported by different existing solutions.

3 GOALS AND REQUIREMENTS

In order to address the previously discussed limitations of existing work we propose a unifying framework that supports the exchange of information across heterogeneous devices and helps developers in avoiding common issues. As such, the framework allows developers to mix and match devices that are not supported in current systems and thereby enables much richer use cases. The framework mainly focusses on addressing issues on the lower technical levels so that it remains versatile enough for developers to plug in their own user interfaces and interaction methods in order to support a wide range of use cases. For instance, our framework might be used to implement a system where users place all kinds of devices on a tabletop and use drag and drop interactions to exchange content, or the framework might be used as the backend for a classic desktop application running on standard consumer hardware. The following goals and requirements focus on creating a solid foundation on the lower level but as detailed in this section they also manifest themselves as solutions for higher-level issues such as those described in the previous section. The three main goals G1–G3 can be summarised as follows:

Goal 1 (G1): Make it easier and faster to develop applications for user-driven information exchange between devices.

Goal 2 (G2): Support the exploration and exchange of information for devices or combinations of devices currently not supported.

Goal 3 (G3): Support developers in providing a consistent user-friendly experience regardless of how different hardware or protocols are combined for a given use case. Further, the framework should not introduce limitations that might restrict the development of customised or novel interfaces and interaction techniques proposed by the HCI community.

Based on the presented related work and by taking special care to avoid shortcomings observed in existing solutions, and by taking into account the issues already identified by the HCI community, we defined a set of requirements for a cross-device information exploration and exchange framework. For instance, from the classification of issues defined by Edwards et al. [7], we avoid *constrained possibilities* via requirement R3 and R8, and *unmediated interaction* as well as *interjected abstractions* via requirement R2 and R5. This in turn contributes to the fulfilment of the previously listed goals.

3.1 User Requirements

The following requirements are related to the users of either the framework (developers) or users of the resulting applications. In most cases the implications for developers also translate into benefits for users interacting with the resulting system.

R1: List, Inspect and Transfer Content. The framework should be able to perform the following three actions in order to build functional applications related to information exploration and exchange (G1): First a user should be able to see a *list* of all the content on a device. Furthermore, it has to be possible to *inspect* specific content in more detail in order to verify that this is the content we are looking for (e.g. check the pages of a PDF document). Finally, the system should make it easy to *transfer* content between two devices. This falls in line with design choices made by other collaborative interfaces such as ActivitySpace [16]. Note that these actions should be supported for units of information with a fixed size as well as continuous data streams of unknown length.

R2: Customisable Yet Consistent User Experience. The current process of transferring content between different types of devices is not always optimal [21, 24]. Depending on the use case and the relevant devices, a different user interface or interaction method might be necessary. We require that developers should be able to integrate their own interfaces or interaction techniques (G3) but the framework should offer the necessary abstraction to allow them to keep the user interface consistent and uniform, regardless of the hardware or technologies that are combined [7, 23].

R3: Reduced Hardware and Software Constraints In order to support a wide range of devices (G2) and provide richer use cases, the hardware requirements demanded by a framework should be as low as possible and devices that might not have a display or I/O modalities should also be supported. There should be no need to install and run specific software on a participating device. One way to achieve this is to make use of the channels already offered by devices instead of having to install software to create new ones. For example, both old and modern mobile phones offer access to contact information and audio via Bluetooth, or some cameras provide access to the pictures stored within via WiFi. Lowering the hardware and software requirements allows the inclusion of many devices that are currently not supported by existing solutions, but a unifying framework has to be designed to cope with new issues introduced by these lowered requirements.

R4: Support for Remote and Non-mobile Devices. In order to not exclude remote or non-mobile devices such as an FTP server (G2), a framework should take into account that devices are not required to be physically present to be accessed. There are different approaches to include such devices. For instance, in the domain of information storage, retrieval and manipulation, virtual representations or tangible physical objects are often used to represent devices that cannot be physically moved [34]. A framework should therefore take into account that devices might have alternative representations in the physical world and in the user interface. This might be important for detecting participating devices or for recognising user-initiated actions.

3.2 Architectural Requirements

Based the goals G1–G3 and the user requirements R1–R4 we can derive additional technical requirements for the architectural level which are independent of the used programming patterns or implementation technologies.

R5: High-level Abstractions In order to support developers (G1) it is important that a framework provides a consistent high-level interface for accessing its functionality. For instance, a developer always has to be able to invoke the list, inspect or transfer actions on devices in the same way, without having to care about the technical differences which are managed by the framework. This allows developers to build more consistent user interfaces (R2) and makes it easier to map specific user interactions to the corresponding actions.

R6: Broad Connectivity and Access Support. In order to integrate new devices (G2) and to deal with requirement R3, it is necessary to support a wide range of communication methods and protocols such as Bluetooth, TCP/IP or WebSockets. Each of these protocols may act as a carrier for other protocols. For example, protocols such as FTP, Samba or RESTful HTTP interfaces are implemented on top of TCP/IP sockets.

R7: Bi-directional Mediation. In order to allow devices with different communication hardware or protocols to communicate (G2, R3), a framework should act as a mediator and translate between different protocols for enabling transfers between radically different types of devices (R1). This requires the framework to not only mediate the gathering (input) but also the writing (output) of information which is not foreseen in various existing mediation frameworks [2].

R8: Extensibility and Reusability. As shown in the background section, existing content exchange approaches are often hard-coded use cases where the participating devices, content types and communication channels are limited and fixed. A truly generic framework should be extensible on all relevant aspects of the process. First of all, there should be no limitations in terms of the hardware that can be used for user input or the UI rendering. In line with requirement R6, it should further be possible to extend the list of supported protocols in order that additional devices can be included while keeping the user interface consistent. Finally, a framework should not be limited in terms of the content types that can be transferred (e.g. images, contacts or music) and be extensible to support new content types. A modular architecture also encourages developers to reuse components, which makes it easier to put together rich use cases (G1).

4 ARCHITECTURE

We now describe the concepts and overall architecture of our unifying INFEX framework for cross-device information exploration and exchange. We motivate our design choices and relate them back to the previously listed requirements R1–R8 to further highlight the benefits of our design with respect to the existing body of work. As motivated earlier, it is not necessary to install any software on the participating devices themselves. Instead, the INFEX framework is set up on a dedicated device such as a Raspberry Pi[1], a laptop or desktop computer. Note that the choice of this dedicated device might also be influenced by the desired user interface ranging from a simple WIMP[2] interface to a more demanding tabletop interface. The INFEX framework uses the dedicated

[1] https://www.raspberrypi.org

[2] "Windows, Icons, Menus and Pointers" interfaces, which are commonly used in desktop environments such as Windows or macOS

device's different communication channels and associated protocols to connect to participating devices. Note that it is also possible for the dedicated device to simultaneously be used as a regular client device, allowing for instance a laptop running the framework to act as a bridge for other devices while also acting as an information source or target. Regardless of its setup, if connected devices use different communication channels and protocols to communicate with INFEX the framework will perform the necessary mediation to unify the devices and connect devices with incompatible hardware or software characteristics. The architecture designed to achieve this is shown in Figure 1.

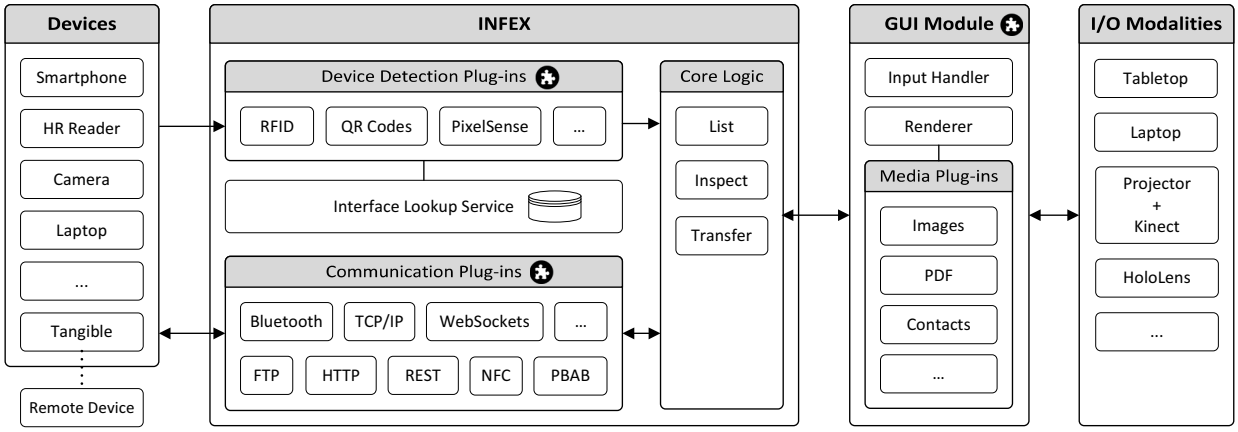

Fig. 1. INFEX architecture

The INFEX architecture consists of multiple components and most of them are extensible or replaceable (R8) as shown in Figure 1. A first important component is responsible for automatically *detecting devices* willing to participate in an information exchange session. Whether explicit or implicit, there will always be a mechanism for the user to specify which devices should be used in the application. Depending on the developed application, a user might include devices by placing a device on an interactive surface, by bringing them into the proximity of a designated device running an INFEX instance or by selecting them from a list in a GUI. Regardless of the chosen mechanism, the framework should be told by the developer how it can detect that a user wants to include a device. Device detection is a fundamental component of the INFEX framework and can be chosen or extended by the application developer via specific plug-ins (R8).

A second major component is responsible for the *communication with devices*. In order to support communication between different kinds of devices, communication protocols and methods are also implemented as plug-ins and our framework mediates the flow of information to connect devices using different communication protocols (R7). The third major component is the GUI component, responsible for *capturing user input* and providing a graphical interface for interactions. An interactive surface might be used to allow users to drag and drop content from one device to another, but due to the interchangeability of the GUI component one can also just use a laptop screen as an interface for information exchange. The core logic of INFEX ties these three components together to support the *listing*, *inspection* and *transfer* of device content (R1).

The presented architecture is language and hardware independent but in Figure 1 some example technologies and devices are shown for each component in order to clarify what might be integrated by developers. The leftmost group of devices labelled `Devices` represents the set of devices that are going to exchange information. The rightmost group of devices labelled `I/O Modalities` includes devices that are used to interact with the system and for instance initialise a data transfer between two devices of the `Devices` group. Depending on the use case, a user might interact with the system

via a tabletop interface, a classic WIMP interface running on a computer or even a virtual or mixed reality interface. In the remainder of this section we provide a detailed description of the different INFEX components coordinating the information exchange and user interaction.

4.1 Device Detection

As mentioned before, the way devices are detected is determined by the used device detection plug-ins. A device detection plug-in is responsible for detecting when devices are joining or leaving a session. A plug-in might also determine a device's position and boundaries which can be used by the GUI to, for example, display an interaction menu next to the device. However, as this is not relevant for all potential interfaces, the spatial detection aspect is optional and only needs to be implemented when relevant. A device detection plug-in is free to use any hardware allowing for many different use cases. Thereby, developers can plug in suitable existing detection frameworks which can, for instance, be based on RFID readers, network discovery or computer vision. The framework also allows more than one detection plug-in to be used at a given time. In order to do so, a developer must specify the set of plug-ins to be used and implement a so-called *detection resolver*. A detection resolver is another kind of pluggable logic that combines the output of the selected detection plug-ins in a way specified by the developer. This way the developer determines what will happen if different plug-ins report conflicting results, or they can use different plug-ins to augment each other. For instance, a plug-in that uses an interactive surface's touch events could be used to detect that a device has been placed on the surface and provide a rough estimate of the location while a second plug-in could use computer vision to identify the device and provide more precise boundaries. Detection plug-ins continuously run in the background allowing them to poll for new devices as well as to notify the framework when a previously connected device leaves the session.

4.2 Device Specification Lookup

In conformance with requirement R3, participating devices do not need to run any specific software. However, a one-time setup of the device is necessary. Nothing needs to be installed on the device itself, but the user should provide the system with some device details before it can participate in any information exchange session. From a technical perspective, one or multiple unique device identifiers must be provided which can be derived by the detection plug-ins when a device is introduced. This can, for example, be the device's Bluetooth MAC address, a unique number embedded in an RFID tag or a 2D barcode attached to the device. More than one identifying number or string can be provided during setup so that identifiers produced by different detection plug-ins can be used to look up the device's specifications even if the plug-ins use significantly different detection methods. Further, the supported media types and communication channels that can be used with the device must be specified so that they can be associated with the previously provided identifiers. For instance, in the case of a smartphone we could specify that a list of contacts is available via the Bluetooth PBAP (Phone Book Access Profile) and that pictures can be retrieved over USB from the filesystem location `/sdcard/DCIM`. A default user interface is provided to simplify this process for the end-user and can be reused in a wide variety of use cases. However, the way users provide information about their devices might depend on how the application is implemented and which interaction hardware is used in the final product. For this reason we also allow developers to create their own interface for this process via a small stand-alone application or as an integrated part of a use case. Custom interfaces for describing devices simply need to send the captured device description to the framework in a specific format where it will added to the list of known devices. The process of describing and adding a new device is arguably one of the biggest usability hurdles for the framework but this is alleviated by allowing the interface to be replaced and improved later.

In the future, a GUI might be provided to compose a configuration by selecting options from a list or one might even automatically probe the device for common interfaces and suggest any detected communication channels. Alternatively, the system could maintain a database of common devices and provide default settings. Once the setup is complete, the device is added to the database and the system will be aware of its presence when an associated identifier is returned by any of the detection plug-ins. In case that multiple plug-ins are used, the detection resolver combines the information and ensures a single identifier is returned together with a position and boundaries. In a next step the INFEX database is consulted to see whether the device is known (i.e. whether it went through the initial setup). If the identifier is not found, the device is simply ignored. However, if the device is known, all device information—including its supported media types and communication channels—is retrieved and passed to the framework's core logic which keeps the information in memory as long as the device is participating in the session.

Note that the object that is used to identify a device does not necessarily have to be the device itself. For instance, one could attach a 2D barcode to a simple everyday object and set it up so that it describes the communication channels to a remote device. Such an object can then be used as a tangible proxy for a device that might not be mobile or located far away. Using the same principle, an interface could allow the user to click an image of the device or select the relevant device name from a menu in the software so that the framework can be told to include the corresponding device based on the device name, not requiring physical interaction with the device itself. This allows interfaces to be more flexible in terms of detection and inclusion mechanisms, but it also provides alternatives for devices where physical interaction is impossible (R4).

4.3 INFEX as Protocol Mediator

One of the major challenges for our INFEX framework was the fulfilment of requirement R7 which requires that we can read content from one device and copy it to another device even if the two devices use completely different communication methods. For this reason the framework needs to be able to communicate via various protocols (R6); preferably in an extensible manner (R8). Therefore, all communication is done via so-called communication plug-ins with each of them implementing a specific communication method such as Bluetooth PBAP, FTP or WebSockets. Note that in order to avoid redundant code, communication plug-ins can make use of other communication plug-ins. For example, a plug-in for RESTful API access could be built on top of an HTTP plug-in which in turn could be built on top of a TCP/IP plug-in.

As mentioned earlier, during the initial device setup each supported media type is specified together with details about its access method. An older phone might be set up to support the sharing of contacts via the Bluetooth PBAP protocol. This way, when contact information needs to be read or written to the phone, the framework knows that it can invoke the Bluetooth PBAP communication plug-in to perform the communication. Each communication plug-in must implement the following operations: `read()`, `readChunk()`, `write()`, `writeChunk()` and `list()`. By defining such a fixed interface, our framework tells each plug-in what to do in the same way, without needing to be aware of their inner workings (R5). These operations on the technical level are sufficient to fulfil requirement R1, allowing developers to provide at least list, inspect and transfer operations in the user interface. The `read()` and `write()` operations allow the framework to read or write a single unit of content (e.g. a document or a contact) while `list()` allows the framework to retrieve a list of all the content that is accessible via the plug-in's communication channel. For transferring content between devices, the framework might use one communication plug-in to read from a device and another communication plug-in to write to a second device. The `readChunk()` and `writeChunk()` operations are variants of the `read()` and `write()` operations specifically for data streams that do not have a clearly defined length. When a data stream is directed from one device

to another the framework repeatedly performs the `readChunk()` operation on the source device and writes the retrieved data to the target device using the `writeChunk()` method.

There is one additional issue that needs to be addressed on the architectural level for both streams and fixed-length content units. The structure of the data retrieved from a device might be different depending on the protocol used to retrieve the data. For instance, contacts retrieved via the Bluetooth PBAP protocol are returned in the vCard format while other communication channels might retrieve contacts as CSV or JSON files. Therefore, a generic data format should be used in the core of the framework and communication plug-ins are then responsible for performing the conversion when reading or writing data. Whenever a communication plug-in is invoked to retrieve data, it transforms the read data into the generic format before handing it to the framework. Similarly, when a plug-in is invoked to write data, data in the generic format is converted to the structure required by the corresponding protocol. The chosen design allows us to isolate protocol-specific complexity into the communication plug-ins in a way that allows INFEX to provide a more generic high-level programming interface. This makes it easier for developers to build richer applications and also blocks developers from letting technical complexity affect the user experience (R2).

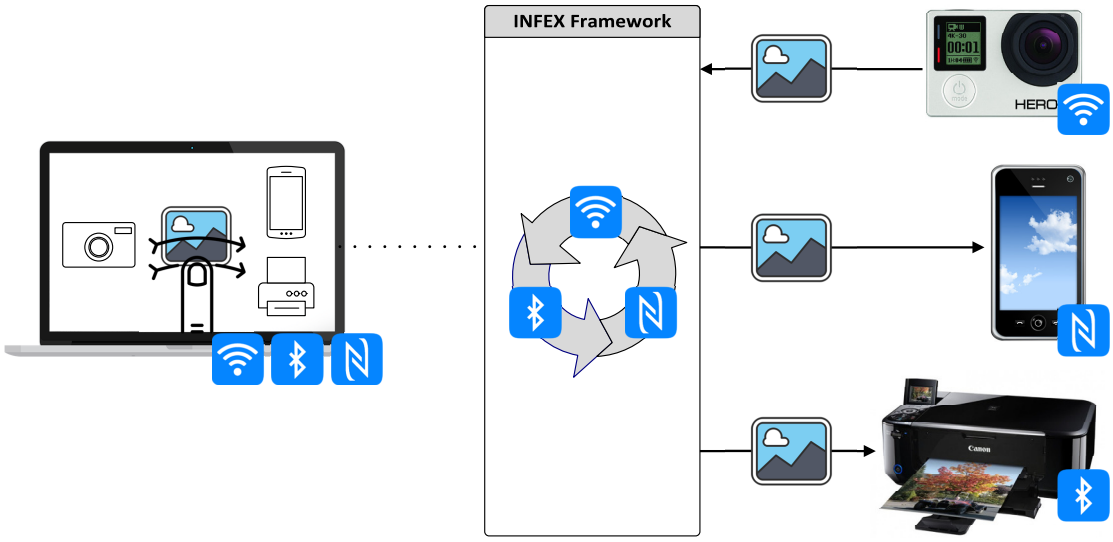

Fig. 2. Example scenario showing how INFEX unifies incompatible devices

In order to illustrate the mediation process, we briefly discuss the scenario shown in Figure 2. In this case the user is running an INFEX-based application where the developer implemented a simple touch-based interface allowing users to simply drag and drop content between icons that represent the detected devices. The user has taken a nice picture with their GoPro action camera and would like to print it on a Bluetooth printer and then copy it to their phone via NFC. Without INFEX the process would be quite tedious, requiring the user to synchronise the GoPro with the computer by using the GoPro software, pairing the printer, opening the image file, manually initiating a print, opening the necessary software for the NFC transfer and manually initiating the transfer. With our INFEX-based application, the user simply drags the image from the GoPro to the printer. The GoPro does not support direct printing nor can it establish a connection with the printer due to the incompatible protocols. In this case, INFEX solves the problem by acting as a mediator and retrieves the image via WiFi from the GoPro in order to then send it to the printer via Bluetooth. However, this is all hidden from the user who experiences the transfer as a simple drag and drop operation. Similarly, the user can drag an image from the GoPro to the mobile phone which will

initiate an NFC transfer if the mobile phone is within range of the laptop's NFC module. Note that the scenario could be extended further and more devices and protocols could be added by the developer without introducing new complexity, allowing all the of the supported devices to be mixed and matched.

4.4 User Interaction with Device Content

So far we have described all the technical components of the architecture and showed how the framework can communicate with different devices. However, in the end all forms of communication and transfer are initiated by the user. Similar to the already described parts of INFEX, also the GUI is not fixed but can be replaced if necessary. As discussed earlier, we should also support devices without a display or input modalities and therefore the GUI module must provide a graphical interface to interact with the detected device. The main goal of the interface is to represent the devices in the session, show what kind of content types a device provides (e.g. contacts or images) and to allow a user to initiate actions such as listing, inspecting or transferring content (R1). As requested by requirement R5, at the data level these actions should be supported in the core of the framework. The requirement further implies that application developers do not need to worry about the technical details of the connection. User interfaces can be developed on top of the *list*, *inspect* and *transfer* abstractions which ensures that developers are offered the necessary technical support while still allowing them to implement highly customised interfaces and interaction techniques (R2). In other words, INFEX provides the infrastructure to perform these actions on the data level but developers can decide how content lists are presented to the end user, how content is inspected and how the user should initiate a transfer. Note that the architecture also makes it easy to use additional hardware such as an interactive surface or depth-sensing cameras as part of the user interface (the group of devices labelled `I/O Modalities` in Figure 1). As discussed later, we provide a default GUI implementation working for most use cases, that can be used with commonly available hardware and which can also serve as a starting point for further extensions.

Concretely, the GUI module should be built in accordance with a specific interface definition in order that bi-directional collaboration with the core logic is possible. This allows the core logic to notify the GUI module when new devices are added or removed to a session as well as when a device's position changes. The GUI module can then ask for the content types that the device supports or directly ask for a list of all content so that it can be visualised accordingly. Similarly, the GUI module can react to captured user input by mapping it on to one of the *list*, *inspect* and *transfer* abstractions and delegating the action to the core logic for execution on the data level. Depending on the use case, GUI modules can be implemented using different technologies. A GUI module might, for instance, be based on web technologies such as HTML, CSS and JavaScript or it could be a WIMP interface built with the .NET framework. Note that the GUI component does not necessarily have to run on the same machine as the rest of INFEX and can communicate with the core over the network using TCP/IP sockets, WebSockets or a RESTful interface.

If a GUI module wants to provide special visualisations or interactions for specific media types, it should implement an internal plug-in system to provide extra functionality for specific content. These media plug-ins may be used to affect the way content of a specific type is listed and inspected. For instance, one might want to visualise a set of images as a grid of thumbnails instead of a list of file names. Similarly, when inspecting a PDF document it can make sense to show a widget that allows the user to go through the pages of the PDF, or an audio widget might allow users to play back any audio files found on a device. Because this behaviour is very content specific, it should be isolated in so-called media plug-ins. As an added benefit, media plug-ins can be reused in GUI modules that use the same technology. For instance, media plug-ins developed in HTML and related technologies for a browser-based interface could also be used in an interface that uses mixed reality since devices

such as the Microsoft HoloLens[3] can integrate HTML-based widgets. Similarly, GUI modules that use technologies such as QML[4] might reuse media plug-ins in compatible environments.

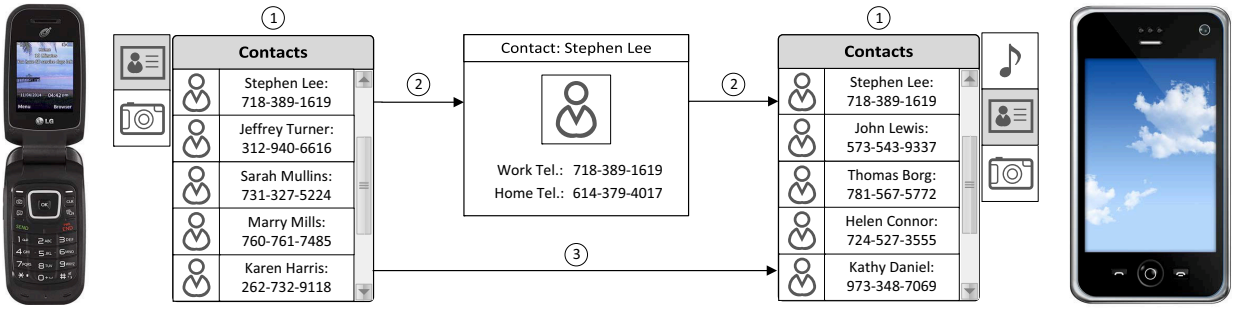

Fig. 3. A possible user interface for listing, inspecting and transferring content based on INFEX abstractions

To outline the benefits of INFEX, we show another possible scenario in Figure 3 where two phones exchange contact information by placing them on an interactive surface. Even though the scenario merely involves the exchange of contact information, the phones would be considered incompatible in existing systems due to the age and hardware difference of the devices. The surface shows a graphical interface next to the phones for user interaction so that users can interact with the phones without having to install any software, which might be impossible for the old phone shown on the left-hand side anyway. The three steps for exchanging information have been marked with circles numbered ① to ③. Device positions returned by the detection plug-ins are used by the GUI module to place an interaction menu next to the devices and a tab is shown for each supported media type. When a tab is selected the *list* abstraction is applied and the returned content is visualised in a list ①. A media plug-in for contacts causes contacts to be listed in a more suitable manner, showing their name, telephone number and a thumbnail image. Furthermore, when a contact is inspected by dragging it onto an empty space on the surface, the *inspect* abstraction is invoked and a specialised widget is shown that visualises the contact in greater detail ②. To initiate the *transfer* abstraction, the user can either drag the inspection widget into the other phone's contact list or a contact can be dragged directly from one list to another list ③. Please keep in mind that this is only one possible implementation and different solutions such as Rekimoto's Pick-and-Drop [25] user interface could also be developed on top of the *list*, *inspect* and *transfer* abstractions. It is also important to note that the developer of the GUI component does not need to know any technical details about the underlying connections as they only have to use the *list*, *inspect* and *transfer* abstractions. For each invocation, INFEX automatically uses the corresponding technology and protocols and converts the data where needed. In this scenario the framework might use Bluetooth to *list* and *inspect* contacts on the older phone on the left, while it might use HTTP to connect to a cloud service to copy the contact to the newer smartphone on the right. Although the scenario is relatively simple it would already take a significant developer effort to create such an application from scratch and the complexity rapidly increases if more devices or media types (e.g. images) have to be supported.

5 IMPLEMENTATION

In this section we present an initial version of the INFEX framework and illustrate how we implemented the previously mentioned requirements and architecture. We start by detailing how we

[3]https://www.microsoft.com/microsoft-hololens/en-us
[4]http://doc.qt.io/qt-5/qmlapplications.html

addressed some of the non-trivial technical issues and then show how the framework can be used by developers.

5.1 Implementation-specific Details

In accordance with the previously mentioned requirements and the proposed architecture, we have implemented a first version of the INFEX framework to demonstrate its potential. The framework is implemented in Java and a classical JAR-based plug-in loading mechanism has been applied to enable the extensible components described in the architecture section. In order to efficiently handle the internal processing of various events (e.g. device detection or the invocation of actions or plug-ins) we make use of Google's `EventBus` implementation forming part of the Guava Google Core Libraries for Java[5]. The `EventBus` offers an efficient publish-subscribe mechanism allowing components of the INFEX framework to publish events to a central pool from where they are automatically redistributed to other components that are relevant to the event. The `EventBus` is, for instance, used by communication plug-ins to notify other components in the system about their progress when performing a transfer. Similarly, when a device detection plug-in detects that a new device has been added to the session, it publishes the detected device identifier in order that the framework can react accordingly. The use of an `EventBus` enables efficient asynchronous event handling and simplifies the orchestration of various components of the framework working in parallel and the event system is further used for error handling and logging purposes.

The extensible components can be divided into three categories of plug-ins: device detection plug-ins, communication plug-ins and GUI modules. Each of these JAR files should implement a plug-in interface by providing a class that implements the interface for the corresponding plug-in type. These JAR-based plug-ins may contain additional dependencies such as libraries or binaries required for accessing hardware relevant to their task. The interface for device detection plug-ins is fairly straightforward. Developers should simply implement an `init()` and a `stop()` method and are further free to implement their detection method as desired. The plug-in is then responsible for publishing the correct events to the previously discussed `EventBus`, for instance when they detect that a device enters or leaves the session, or optionally when spatial information changes (e.g. the device's location or boundaries). When these events are published they will be captured by the session manager in the core and the corresponding internal actions will be executed. For instance, when a `KnownDeviceDetected` event is published the framework looks up the device's specifications and notifies the GUI module about the new device in the session. Note that if a detection resolver was provided, the framework will give the last result of each active detection plug-in to the resolver which is then responsible for combining the results and publishing the relevant events.

Communication plug-ins have to implement the `read()`, `readChunk()`, `write()`, `writeChunk()` and `list()` main methods. As mentioned earlier, the `readChunk()` and `writeChunk()` methods are variants of the `read()` and `write()` methods that deal with data streams of unknown length and will be called repeatedly by the framework to deal with potentially endless streams. When any of the methods are called, a HashMap containing key/value pairs with additional parameters is passed along. For instance, if a device states that its images can be retrieved via FTP from a particular folder, the IP address and folder path need to be given to the FTP plug-in when it is invoked to list content. Since INFEX keeps track of all the devices and their communication specifications, it is able to automatically provide the relevant parameters to the corresponding plug-ins when the user triggers an action. An example of a communication plug-in is described later in Section 5.2. Note

[5]https://github.com/google/guava

that although a communication plug-in has to implement all the mentioned methods, in some cases this might not be possible. For instance, not all communication plug-ins can read or write data as a stream (e.g. the Bluetooth PBAP protocol only deals with phone contacts as fixed-sized content units). There are also devices that only *produce* digital data (e.g. scanners or webcams) or devices that only *consume* digital data (e.g. printers, speakers or televisions). Therefore, plug-ins for the corresponding protocols cannot always provide both read and write methods at the same time. In such cases all the methods still have to be implemented, but the unsupported ones should fire an `UnsupportedCommunicationPluginMethod` event when invoked so that the framework can deal with the situation accordingly and notify the GUI module in order that the incompatibility can be visualised to the user. Figure 4 shows a scenario where streams as well as consume-only and produce-only devices are combined in a particular application. In this case the diagram shows how INFEX can retrieve information from a scanner (produce-only device) and visualise the resulting digital document on a television screen (consumer-only device) by, for example, implementing a communication plug-in for the Chromecast[6] protocol. The same plug-in would also allow audio to be streamed to the television from a Bluetooth device such as a sports watch. The user interface that allows the user to initiate these exchanges is of course flexible and in this case a tabletop is used, potentially with tangible objects to represent devices such as the scanner and the television.

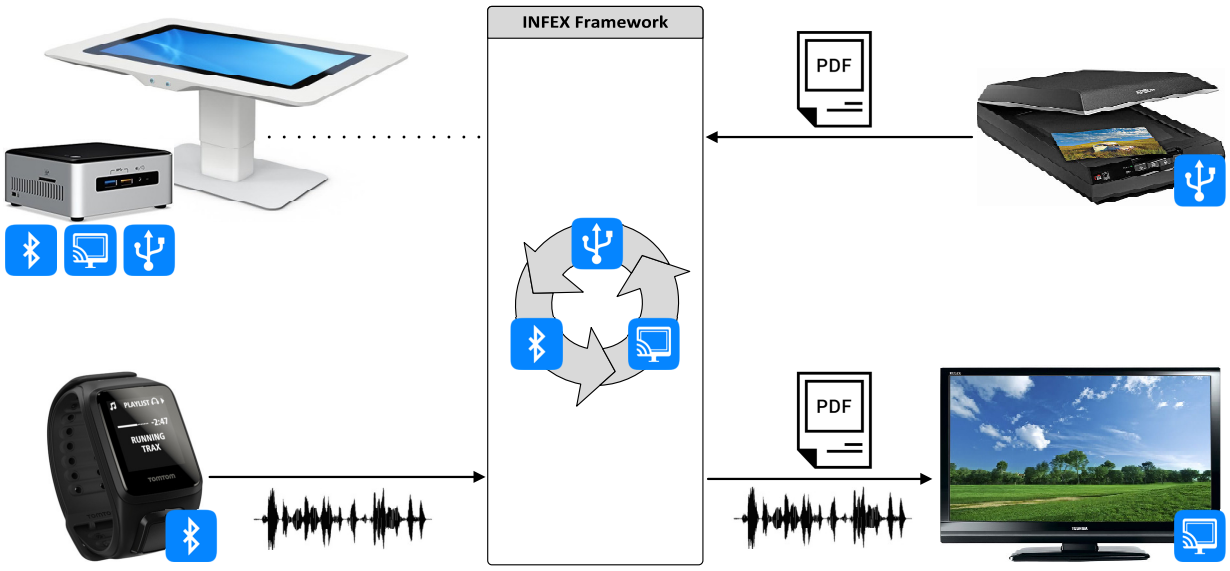

Fig. 4. Example scenario showing a particular application of stream mediation and transfers between consume-only and produce-only devices

As discussed in the architecture section, the structure of retrieved information sometimes depends on how the data was retrieved. For instance, when using the Bluetooth PBAP protocol contacts are returned in the vCard format shown in Figure 5a while Google's cloud storage for contacts will return contacts as XML format as illustrated in Figure 5c. These formats are incompatible and even though the framework can mediate the corresponding data transmission protocols, exchanging contacts between these storage media would not be possible without conversion. In order to address this issue, it is necessary that a communication plug-in translates any retrieved data to a generic representation before passing it on. This generic representation is then used internally, and is eventually passed on to another communication plug-in which will convert it to the target device's data format for writing.

[6]https://www.google.com/chromecast/

```
BEGIN:VCARD
VERSION:2.1
N:Doe;John;;;
FN:John Doe
TEL;CELL:+12025550196
EMAIL;PREF;HOME:jd@mail.com
END:VCARD
```

(a)

```
{
 "type": "contact",
 "data": {
  "displayName": "John Doe",
  "fullName": ["John", "Doe"]
  "phone": ["+12025550196"],
  "email": ["jd@mail.com"]
 }
}
```

(b)

```
<atom:entry xmlns:atom="..." xmlns:gd="...">
  ...
  <gd:name>
    <gd:givenName>John</gd:givenName>
    <gd:familyName>Doe</gd:familyName>
  </gd:name>
  <gd:email rel="..." address="jd@mail.com"/>
  ...
</atom:entry>
```

(c)

Fig. 5. (a) Contact data as retrieved by the PBAP plug-in (b) Contact data converted to the generic format for internal usage (c) Contact data converted to Google's format by the Google Contacts plug-in

Similarly, the write method implemented by a communication plug-in will be passed data in the generic format and then convert or interpret it according to the specifics of the implemented protocol. Therefore, the write method of a communication plug-in does not need to know where the data came from or how it was retrieved; it only needs to know how to convert the generic format to the format required by the target device ensuring compatibility with any other communication plug-in that can generate data of that type. In our implementation JSON is used for internal information representation. Each object stores the original content type (e.g. contact, image, movie or PDF document) together with a set of key-value pairs that represent or link to the relevant information. For example, the vCard data shown in Figure 5a is converted to the JSON format shown in Figure 5b for internal usage and then converted to the XML-based data format shown in Figure 5c if it is to be written to a device via Google's Contacts API.

In order to standardise the internal representation and to help plug-ins handle them, each internal data object is tagged with with a media descriptor (much like a MIME tag) describing its content (e.g. "`file/image/png`". Just like MIME types media descriptors represent a hierarchy of types and subtypes describing the content, with the difference that we allow for more than two levels. The structure of each data type is defined in JSON files, in the form of a list of fields and their data types (e.g. String, Integer, FilePath or ByteArray). These format definitions can be added or extended by developers. As expected, each subtype inherits the structure of the supertype which might be extended with additional fields. The benefit of this representation is that communication plug-ins can make use of the fact that they know the subtype, but they do not have to and can also treat the object based on its basetype. For instance, an FTP communication plug-in that receives an object of the type "`file/image/png`" can ignore the fact that it is an image with PNG encoding and will just address the document as any regular file (e.g. reading and writing it as a byte array). On the other hand, another device might only accept bitmap images and the corresponding communication plug-in might then use the details related to the image encoding so that it can apply the correct conversion before writing it as a bitmap to the device.

In our current version, one GUI module has been implemented but special care has been taken to make this default GUI module as generic as possible. This ensures that developers have a working starting point and only need to write a new GUI module if they have special requirements. The implemented GUI module is based on web technologies including HTML5, JavaScript and CSS which allows the GUI to run on any interface hardware that can run a modern web browser (e.g. a dedicated laptop or a tabletop). The GUI module contains a lightweight web server for serving the static web files that implement most of the visualisation and interaction logic. Once the web files are delivered to the web browser, JavaScript is used to open a WebSocket connection to the GUI module and connects the HTML-based front end with the GUI module. This bidirectional communication is needed to, for example, notify the front end when a new device has been detected in order that it can display the necessary UI elements next to it. Similarly, when the user performs

an action on the front end such as dragging content from one device to another, the requested action is passed back to the GUI module that invokes the request handler to perform the action.

As discussed earlier, the GUI module can also provide a modular mechanism for visualising specific content types while listing or inspecting content. In this case the media plug-ins are implemented in JavaScript and CSS. When the front end needs to display a list of content of a specific type, the plug-in for that type is given the list of content and is responsible for generating the HTML code that will be injected into the DOM tree. The widgets for inspecting content are generated by the plug-ins in the same way. Note that this default HTML-based GUI module might be replaced to suit the specific needs of both the application developer and the end user.

5.2 Usage

As detailed earlier, client devices do not need to install specific software but instead a dedicated device is set up to unify the connected client devices. Even if the INFEX framework runs on the dedicated device, it does not prevent the dedicated device from participating as an ordinary client device. Since the INFEX framework is implemented in Java, the dedicated device needs to be able to run a Java Virtual Machine but apart from than there are no other requirements. However, in order for INFEX to mediate protocols such as Bluetooth or NFC, it is necessary that the dedicated device contains the necessary hardware. For this reason a small form factor computer such as a Raspberry Pi might be sufficient for simple use cases, but a laptop or desktop computer with additional communication hardware is preferred for more complex use cases.

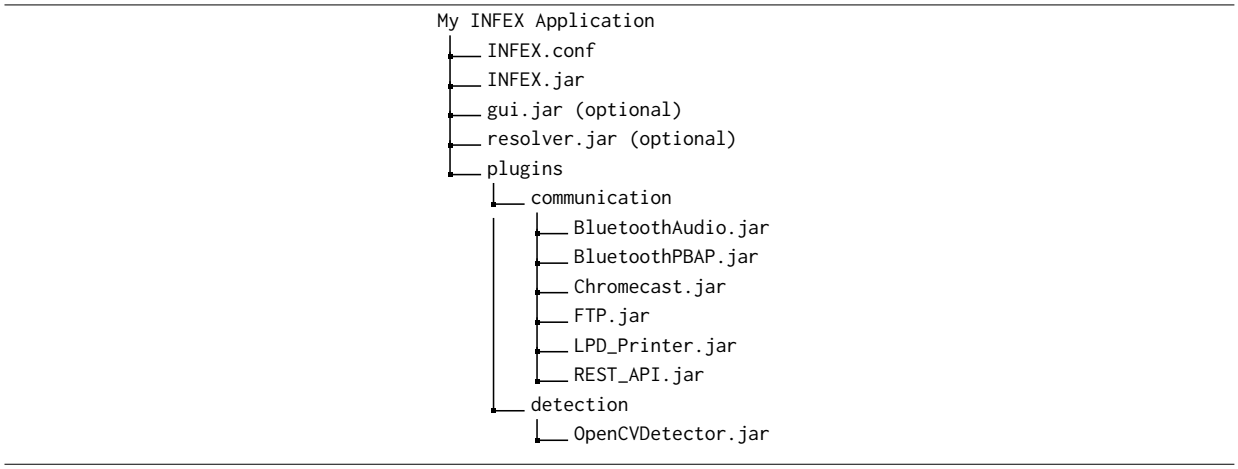

Fig. 6. Example deployment of the INFEX framework

The INFEX framework comes as a JAR file and can be used a standalone server providing access to the functionality via WebSockets or TCP/IP, or the JAR file can be used as a standard library in another application providing direct access to the functionality. In some cases it may be desirable to deploy INFEX as a standalone server as it allows the user interface to run on another device and communicate with the INFEX instance remotely. Regardless of how the framework is used, additional files such as plug-ins should be placed in predefined directories next to the main JAR file. The default detection resolver and the GUI module can be replaced by placing plug-ins named `resolver.jar` and `gui.jar` in the same directory as the main JAR. Figure 6 shows an example deployment of the framework.

Depending on the use case, the framework might need to mediate different protocols by means of communication plug-ins. A plug-in should be provided for each protocol that needs to be supported, but these plug-ins can of course be shared and reused by developers. To implement a communication

plug-in, the developer must implement the `read()`, `readChunk()`, `write()`, `writeChunk()` and `list()` methods of the `CommunicationPlugin` plug-in interface.

Listing 1 shows an example of how the principles explained in Section 4 have been concretely implemented for an FTP plug-in based on the Apache Commons Net library. The methods to read, write and list content on an FTP server are implemented (with the help of an external library) and the event system is used to publish the relevant events so that the core framework can use the plug-in in the mediation process without further intervention from the developer. The plug-in should be compiled into a JAR file and placed in the plug-in directory for communication plug-ins. INFEX will then be able to automatically invoke these methods and use them in the mediation process when that specific protocol is encountered.

```
1  public class FTPCommunicationPlugin implements CommunicationPlugin {
2
3   public static final String protocolName = "FTP";
4
5   public ContentUnit read(int taskID, Device device, CommunicationConfig config, String path) {
6    PubSub.publish(new CommunicationReadStartEvent(this, taskID, device.getID(), path));
7    ByteArrayOutputStream os = new ByteArrayOutputStream();
8    FTPClient ftp = new FTPClient();
9    ftp.connect(config.get("server"), Integer.parseInt(config.get("port")));
10   ftp.login(config.get("username"), config.get("password"));
11   PubSub.publish(new ComPluginReadProgressEvent(this, 0, taskID, device.getID(), result.path);
12   ftp.retrieveFile(path, os); // intermediate progress events removed to keep the example brief
13   PubSub.publish(new ComPluginReadProgressEvent(this, 100, taskID, device.getID(), result.path);
14   ContentUnit result = new ContentUnit(INFEX.FileTypes.GENERIC_FILE);
15   result.setField('path', path);
16   result.setField('data', os.toByteArray());
17   result.setField('contentsize', result.getField('data').length);
18   PubSub.publish(new ComPluginReadEndEvent(this, taskID, device.getID(), result));
19   return result;
20  }
21
22  public ContentUnit readChunk(int taskID, Device device, CommunicationConfig config, String path){
23   PubSub.publish(new UnsupportedComPluginMethod(this, CommunicationMethods.readChunk));
24   return null;
25  }
26
27  // write and writeChunk methods are similar to the read and readChunk methods and are left out
28
29  public ContentUnit[] list(int taskID, Device device, CommunicationConfig conf, String path) {
30   ... // similar to the read method but returns a array of ContentUnit objects containing
31   ... // the file metadata but without the byte content
32  }
33
34 }
```

Listing 1. Implementation of an FTP communication plug-in

Detection plug-ins are also implemented according to a plug-in interface and compiled into JAR files. Detection plug-ins are initialised once at startup and are then free to execute whatever logic needed to detect devices continuously. As an example, a plug-in could be developed to scan the network for devices, it could use computer vision to detect devices placed on a specific surface or it might simply listen for events originating from the user interface (e.g. a user selects a device from a list to be added). When the plug-in detects that devices are added or removed the plug-in should broadcast the corresponding events (e.g. `KnownDeviceDetected` or `KnownDeviceLeft` events) so that INFEX can update the session state.

When the INFEX framework is run, either as a standalone server or embedded in an application as a library, participating devices should be introduced to the framework in a one-time setup process. A default interface is provided to do this, but developers may also implement their own

to gather device specifications and send it to the framework. Figure 7 shows one of the panels offered by the default setup wizard. A list of supported protocols and their relevant details (e.g. an IP address or user credentials) should be provided, and based on a plug-in's `protocolName` field the framework will know which plug-in to invoke when mediation is needed for that protocol.

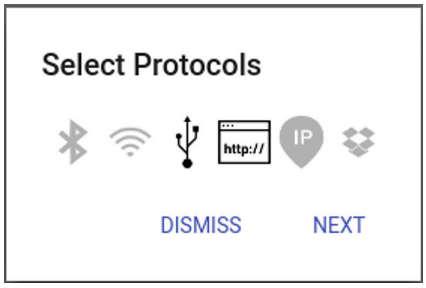

Fig. 7. Device setup wizard

After adding devices, the device detection plug-in keeps track of devices in the session and INFEX will keep active devices' specifications (e.g. supported protocols and media types) in memory. Applications that use INFEX are kept up to date when devices are added or removed in the session and also their specifications are shared so that the user interface may visualise their capabilities accordingly. Applications that build upon INFEX are then free to initiate content listings or transfers, for instance based on a user interaction in the user interface. As the application is aware of the devices' capabilities, it knows what protocols are relevant when a user initiates a transfer. The results are returned to the application which can then handle them accordingly (e.g. visualise them in a list). How users initiate a transfer depends on how the application is implemented but if a transfer is invoked by the user, it can again ask INFEX to perform the transfer by specifying the source device, protocol, content path as well as the target device, protocol and target path. The transfer is then handled completely by INFEX which will perform the necessary mediation and keep the application updated about the progress.

Please keep in mind that a default GUI module is provided and therefore a generic INFEX-based application could simply be assembled from existing detection and communication plug-ins without writing any additional code. The development of additional plug-ins, GUI modules or client applications is only needed to add support for additional protocols or for customised user interfaces.

6 USE CASE

As an infrastructure framework, INFEX allows developers to unify devices with different hardware characteristics within a single application. This allows developers to integrate devices that have been overlooked so far and to create new interesting use cases that would be far from trivial when implemented from scratch. As INFEX focusses on the lower technical issues associated with information exchange, it makes little sense to evaluate INFEX from an end-user perspective. We rather provide a technical evaluation of the INFEX framework by detailing an INFEX-based application for exploring and exchanging content on various devices, including some previously unsupported ones. In order to verify our architectural choices, we have built the necessary plug-ins to replicate and extend some common use cases introduced earlier in the background section. We also introduce some new use cases and unconventional devices and show how they can be combined with existing use cases, illustrating the unification capabilities of our framework. Note that the presented combination of devices and media types has not been integrated as a working whole in any of the existing solutions before and it is only one example of what is possible with

the INFEX framework. Further, it is interesting to point out that existing tools and use cases are often limited and cannot be easily extended, making it hard if not impossible to achieve the same results with existing frameworks. Of course, a similar use case can always be built from scratch but we intend to show that INFEX already offers most of the functionality that would have to be implemented. More importantly, INFEX-based applications can later be extended and reconfigured as this usually involves the addition or removal of a plug-in rather than having to make changes throughout the whole application. The offered abstractions also encourages developers to keep their user interface consistent and uniform, which helps to prevent user experience issues that can surface due to underlying technical complexity.

For the presented example interface, we have been inspired by systems such as ActivitySpace [16] and it is therefore based on an interactive tabletop where participating devices are placed on the surface. However, we must stress again that is only one possible way of using INFEX and developers are free to use any kind of user interface or interaction hardware. The same application could easily be modified to use network scanning for detection and a point-and-click user interface on a standard laptop. For this particular application, our default HTML-based GUI module was used. Detection plug-ins based on touch and computer vision are used to detect devices and estimate their position on the interactive surface. Once detected, the supported media types are queried and an interaction menu is shown next to the device, with a tab for each supported media type. Selecting a tab lists all content of the associated type. Dragging an item to an empty space on the surface opens a widget for inspecting that item, and dragging either a list item or a widget onto another device's content list initiates a transfer if the media types are compatible. Figure 8 shows an INFEX session in progress where most of the features mentioned in this section are used.

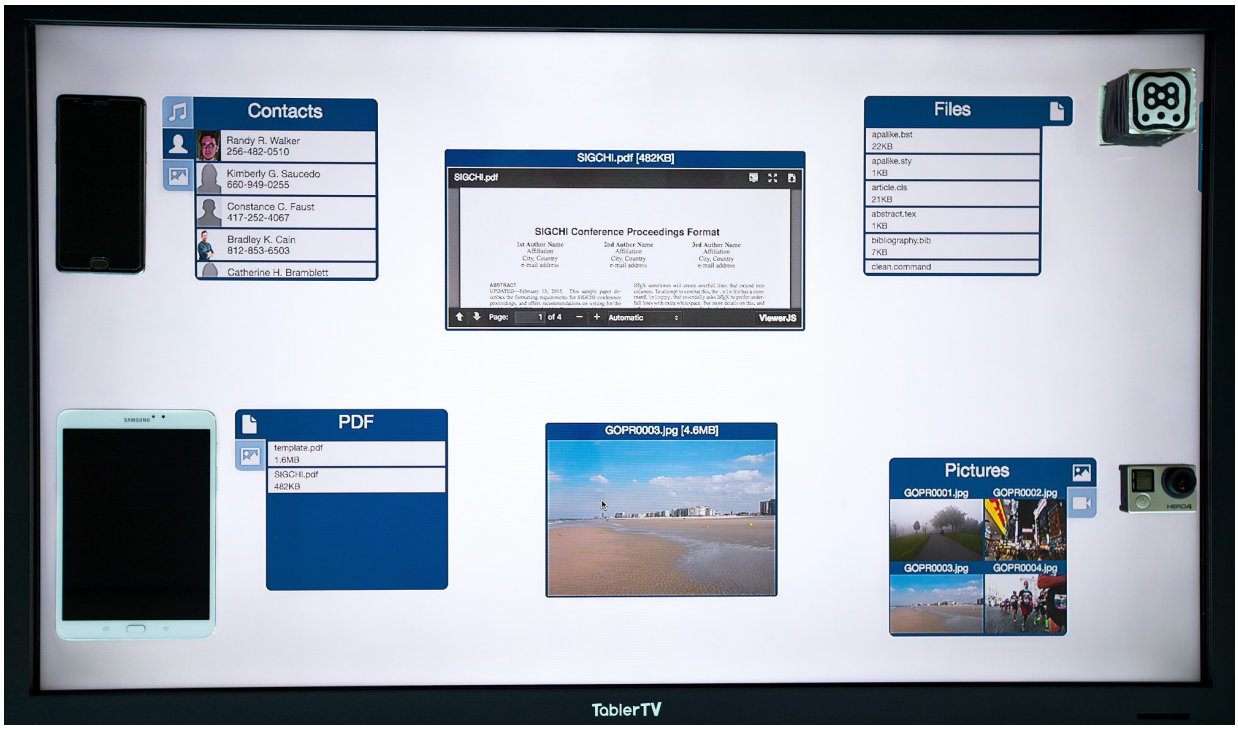

Fig. 8. INFEX application with tabletop-based cross-device information exploration and exchange

In the background section we have seen that one of the most common use cases is the exchange of pictures between two smartphones or between a smartphone and another device [5, 6, 20, 28, 32]. In order to replicate this use case, we have created the plug-ins needed to exchange pictures between an

Android smartphone shown in the upper-left corner of Figure 8 and a GoPro action camera shown in the lower-right corner of Figure 8. To access the photos on the phone we use a communication plug-in that uses the Google Photos API. Alternatively the phone could be connected to the system via an USB connection and a file system connection plug-in could be used for accessing the DCIM folder. In order to access pictures on the second device (GoPro), we used the HTTP protocol over a local WiFi connection to read, write and list content stored on the device. Since both devices can now be accessed by the framework via communication plug-ins, pictures can be dragged out of a device for inspection or can be transferred between the two normally incompatible devices. Specialised media plug-ins are used to show the list of images as a grid and for viewing images in a widget. In order to show the extensibility of the communication protocols, we also mixed in a use case that uses Bluetooth to access a smartphone's contact list via the Bluetooth PBAP protocol and visualise it via the contact media visualisation plug-in. Furthermore, the architectural choices behind the framework make it relatively easy to use tangible objects as physical interfaces to remote content sources. We demonstrate the use of tangibles by replicating a scenario similar to the ones presented in [14, 33]. In our case a physical object (cube) with a fiducial marker on its top is used as a proxy for a remote FTP server as shown in the upper right part of Figure 8. Images from the GoPro can for example be uploaded to the FTP server with a simple drag and drop interaction from the list of images shown next on the side the GoPro to the list of files shown next to the tangible object representing the FTP server. Finally, in order to demonstrate the extensibility of the supported media types and their visualisation, we have implemented a media plug-in for PDF documents. A media plug-in does not only influence how a content type is listed, but also determines how a media type is inspected when it is dragged to an empty space on the surface. For the implementation of our PDF inspection widget this implies that a user can interactively explore PDF documents on the table. In Figure 8, a PDF document has been dragged from the list of PDF files shown next to the tablet in the lower left corner and dropped to the centre of the tabletop for further detailed inspection. Once the user is sure that this is the document they are looking for, the document could, for instance, also be dragged to the list of files next to the tangible in the upper-right corner which will upload the file to the associated FTP server.

Fig. 9. 3D-printed tangibles (with unique touch point patterns at the bottom) representing a television, a webcam and a printer

We have then further extended the example application and have added support for devices that are not usually seen in related work. For instance, a Chromecast plug-in was implemented which allows images or a PDF document to be dragged from any device to a 3D-printed television tangible. This shows the content on the associated television as expected. We have also created a plug-in to support simple webcams. It was implemented so that both still images (via the *read* method) as well as real-time video streams (via the *readChunk* method) can be read from the camera and transferred to other devices. This allows users to for instance capture a video fragment and write

the result to any other device. The stream can also be "written" to the television which shows the video feed on the television in real-time. Plug-ins were also written to support network printers and scanners which offers many more interesting device transfers. Small 3D-printed tangibles were used to represent these devices on the tabletop and can be seen in Figure 9. As for the functionality, one can for instance drag content from the scanner to the television to show the physical document on the scanner plate on the television. Similarly, one can also drag an image from the webcam to the printer to get an immediate print-out. We would like to point out however that the architectural design makes any two plug-ins compatible on the technical level and many more interactions between the use case's devices are possible out of the box. This implies that when developers add a plug-in for a single protocol that it can directly be used in combination with any other plug-in and developers can write the code for the protocol in isolation without taking other protocols and combinations into account.

7 DISCUSSION AND FUTURE WORK

We witness an increasing number of digital devices in our lives, but as illustrated in the introduction, it is not always trivial to retrieve or exchange content between different devices. Existing scenarios for user-driven content exchange place strict limitations in terms of the hardware or suffer from shortcomings on the usability level. This is often the result of the technical complexity and incompatibility which propagates all the way up to the user experience, an issue that has been previously described by the HCI community [7].

We have introduced the INFEX framework which implements the functionality needed to transfer content between heterogeneous devices. Architectural design choices make sure that the framework is highly modular allowing it to be deployed for a wide variety of use cases. In order to illustrate how the INFEX framework improves upon existing work, we refer back to the limitations of existing solutions described in Section 2.4 and again use the terminology defined by Edwards et al. [7].

The issue of *Constrained Possibilities* is addressed by making the framework highly modular and extensible. INFEX removes most of the hardware and software restrictions seen in existing solutions. For instance, participating client devices are not required to install additional software (which is not always possible) and the design of the framework facilitates the use of devices without a display or input modalities. Furthermore, support for new protocols (and thus new devices) is easily added via communication plug-ins allowing developers to mix-and-match devices that are currently not considered in existing work due to technical complexity. Finally, our framework does not enforce any particular user interface or user interaction hardware allowing developers to create customised user experiences ranging from a simple WIMP-based interface to more innovative interactions using state-of-the-art hardware.

Unmediated Interaction is approached by letting the framework handle the technical complexity associated with information exchange. Developers are offered high-level abstractions to transfer content between devices and INFEX isolates protocol-specific details in their respective plug-ins. The other way around, developers can develop communication plug-ins without having to take other protocols into account, as the framework will automatically act as a mediator for exchanges between plug-ins. By offering high-level abstractions, we ensure that developers do not let any of the low-level technical complexity show up in the user interface. This encourages the creation of a uniform and consistent user interface regardless of the mix of devices and protocols that are used. Note that this is in contrast with many existing solutions where interactions might be different depending on the device or protocol, simply because the developers push some technical complexity to the end-user instead of dealing with it internally.

Finally, the problem of *Interjected Abstractions* is also avoided by architectural design choices. Abstractions defined by the INFEX framework do not result in limitations in terms of the supported

devices or user interface. For example, in contrast to existing frameworks or protocols client devices are not required to support one specific programming language or protocol which is the case with the RPC or the UPnP protocol. Furthermore, user interfaces made on top of the INFEX framework are not required to be written in the same language and do not even have to run on the same machine.

By addressing these issues, we achieved our goals G1–G3 described in Section 3. Adding support for new devices and protocols to an existing system would normally become more complex with every added protocol which is why developers often limit support to a set of devices with strict requirements. INFEX deals with most of the technical complexity and ensures that the effort required to add an additional protocol does not increase. As more developers are going to use INFEX, more communication plug-ins can be shared and reused by applications, thereby reducing the effort to build new applications. We would again like to point out that developers can create their own highly custom user interfaces on top of the INFEX framework. We have shown an example of a graphical interface running on regular laptops as well as tabletop-based interfaces making use of tangibles. In the future, virtual or augmented reality technologies might be used to build novel interfaces. It is interesting to point out that we were able to use INFEX to replicate, extend and combine many of the interactions presented in Section 2 without having to start over for every use case. This indicates that INFEX might be valuable for the research community as a rapid prototyping platform for novel interaction techniques, resulting in richer use cases across a wider range of devices.

We see two minor aspects which might currently affect the wider spread of the INFEX framework. First, in order to allow devices to participate without installing dedicated software, an initial setup is always required which could potentially be troublesome for some users. Although a user-friendly setup wizard can make this step easier, users still need to have some technical knowledge about the device they are going to add. Second, there is a minority of devices that can currently not be integrated in INFEX-based applications. While most devices make use of standardised protocols which are easy to support in INFEX, some devices only support proprietary protocols or communication hardware. In a number of cases it might be possible to reverse engineer such a protocol but for some devices it might be impossible to add support due to encryption of the content or protocol.

The implementation of the use case reveals a number of aspects that might be improved in a future version of INFEX. First of all, extensibility is currently achieved via a simple JAR-based plug-in mechanism. While this was sufficient for an initial evaluation of our architecture, a more robust and developer-friendly approach based on the OSGi[7] dynamic module system for Java might be investigated. The current architecture also assumes that INFEX-based applications run in isolation. As part of future work we plan to evaluate the framework with software developers in order to further validate the framework. We also intend to explore the potential of allowing different INFEX instances to communicate with each other over the Internet. This would allow users to use their local instance of an INFEX application to exchange content with devices connected to another INFEX application running at another location. Next, we are also considering operations that do more than just copying data such as cut or delete operations. However, this introduces new challenges related to user permissions and security that require further investigation. Finally, we also plan to make INFEX available to other researchers working on cross-device information exploration and exchange.

[7]https://www.osgi.org

8 CONCLUSION

We have presented INFEX, a unifying extensible framework for cross-device information exploration and exchange. INFEX allows developers to exchange information between devices that are not supported in existing solutions while at the same time ensuring that the increased heterogeneity does not negatively affect the user experience. In contrast to related work, INFEX does not require any special software to be installed on participating devices and takes into account that devices might not have a display or other I/O modalities. By acting as a mediator, INFEX automatically handles content retrieval and transfer. It automatically translates between protocols and data formats to connect arbitrary devices. Last but not least, the extensibility and reusability of plug-in components increases hardware support and allows developers and researchers to focus on the creation of interesting new use cases and novel interaction techniques for cross-device information exploration and exchange.

REFERENCES

[1] Monica Anderson. 2015. Technology Device Ownership: 2015. http://www.pewinternet.org/2015/10/29/technology-device-ownership-2015

[2] Pierre-Alain Avouac, Philippe Lalanda, and Laurence Nigay. 2012. Autonomic Management of Multimodal Interaction: DynaMo in action. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2012)*. Copenhagen, Denmark. https://doi.org/10.1145/2305484.2305493

[3] Jakob E. Bardram. 2005. The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*. Kauai Island, USA. https://doi.org/10.1007/11428572_7

[4] Jacob T. Biehl, William T. Baker, Brian P. Bailey, Desney S. Tan, Kori M. Inkpen, and Mary Czerwinski. 2008. Impromptu: A New Interaction Framework for Supporting Collaboration in Multiple Display Environments and its Field Evaluation for Co-located Software Development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2008)*. Florence, Italy. https://doi.org/10.1145/1357054.1357200

[5] Raimund Dachselt and Robert Buchholz. 2009. Natural Throw and Tilt Interaction Between Mobile Phones and Distant Displays. In *CHI 2009 Extended Abstracts on Human Factors in Computing Systems (CHI EA 2009)*. Boston USA, 3253–3258. https://doi.org/10.1145/1520340.1520467

[6] Andreas Dippon, Norbert Wiedermann, and Gudrun Klinker. 2012. Seamless Integration of Mobile Devices into Interactive Surface Environments. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces (ITS 2012)*. Cambridge, USA, 331–334. https://doi.org/10.1145/2396636.2396693

[7] W. Keith Edwards, Mark W. Newman, and Erika Shehan Poole. 2010. The Infrastructure Problem in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2010)*. Atlanta, USA. https://doi.org/10.1145/1753326.1753390

[8] Luca Frosini, Marco Manca, and Fabio Paternò. 2013. A Framework for the Development of Distributed Interactive Applications. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2013)*. London, UK, 249–254. https://doi.org/10.1145/2494603.2480328

[9] Issac Garcia, Gabriel Pedraza, Bassem Debbabi, Philippe Lalanda, and Catherine Hamon. 2010. Towards a Service Mediation Framework for Dynamic Applications. In *Proceedings of the IEEE Asia-Pacific Services Computing Conference ((APSCC 2010)*. Hangzhou, China. https://doi.org/10.1109/APSCC.2010.90

[10] Hans Gellersen, Carl Fischer, Dominique Guinard, Roswitha Gostner, Gerd Kortuem, Christian Kray, Enrico Rukzio, and Sara Streng. 2009. Supporting Device Discovery and Spontaneous Interaction with Spatial References. *Personal Ubiquitous Computing* 13, 4 (May 2009), 255–264. https://doi.org/10.1007/s00779-008-0206-3

[11] Saul Greenberg, Nicolai Marquardt, Till Ballendat, Rob Diaz-Marino, and Miaosen Wang. 2011. Proxemic Interactions: The New Ubicomp? *interactions* 18, 1 (January 2011), 42–50. https://doi.org/10.1145/1897239.1897250

[12] Peter Hamilton and Daniel J. Wigdor. 2014. Conductor: Enabling and Understanding Cross-device Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2014)*. Toronto, Canada, 2773–2782. https://doi.org/10.1145/2556288.2557170

[13] Mike Hazas, Christian Kray, Hans Gellersen, Henoc Agbota, Gerd Kortuem, and Albert Krohn. 2005. A Relative Positioning System for Co-located Mobile Devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys 2005)*. Seattle, USA, 177–190. https://doi.org/10.1145/1067170.1067190

[14] Simo Hosio, Fahim Kawsar, Jukka Riekki, and Tatsuo Nakajima. 2007. DroPicks - A Tool for Collaborative Content Sharing Exploiting Everyday Artefacts. In *Ubiquitous Computing Systems*, Haruhisa Ichikawa, We-Duke Cho, Ichiro

Satoh, and HeeYong Youn (Eds.). Lecture Notes in Computer Science, Vol. 4836. 258–265. https://doi.org/10.1007/978-3-540-76772-5_20

[15] Steven Houben and Nicolai Marquardt. 2015. WatchConnect: A Toolkit for Prototyping Smartwatch-Centric Cross-Device Applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI 2015)*. Seoul, Republic of Korea, 1247–1256. https://doi.org/10.1145/2702123.2702215

[16] Steven Houben, Paolo Tell, and Jakob E. Bardram. 2014. ActivitySpace: Managing Device Ecologies in an Activity-Centric Configuration Space. In *Proceedings of the 9th ACM International Conference on Interactive Tabletops and Surfaces (ITS 2014)*. Dresden, Germany. https://doi.org/10.1145/2669485.2669493

[17] Shahram Izadi, Harry Brignull, Tom Rodden, Yvonne Rogers, and Mia Underwood. 2003. Dynamo: A Public Interactive Surface Supporting the Cooperative Sharing and Exchange of Media. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST 2003)*. Vancouver, Canada. https://doi.org/10.1145/964696.964714

[18] Clemens Nylandsted Klokmose, Janus Bager Kristensen, Rolf Bagge, and Kim Halskov. 2014. BullsEye: High-Precision Fiducial Tracking for Table-based Tangible Interaction. In *Proceedings of the 9th ACM International Conference on Interactive Tabletops and Surfaces (ITS 2014)*. Dresden, Germany, 269–278. https://doi.org/2669485.2669503

[19] Ming Li and Leif Kobbelt. 2012. Dynamic Tiling Display: Building an Interactive Display Surface Using Multiple Mobile Devices. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia (MUM 2012)*. Ulm, Germany, Article 24, 24:1–24:4 pages. https://doi.org/10.1145/2406367.2406397

[20] Andrés Lucero, Jussi Holopainen, and Tero Jokela. 2011. Pass-them-around: Collaborative Use of Mobile Phones for Photo Sharing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2011)*. Vancouver, Canada, 1787–1796. https://doi.org/10.1145/1978942.1979201

[21] Nicolai Marquardt, Till Ballendat, Sebastian Boring, Saul Greenberg, and Ken Hinckley. 2012. Gradual Engagement: Facilitating Information Exchange Between Digital Devices As a Function of Proximity. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces (ITS 2012)*. Cambridge, USA, 31–40. https://doi.org/10.1145/2396636.2396642

[22] Nicolai Marquardt, Ken Hinckley, and Saul Greenberg. 2012. Cross-Device Interaction via Micro-mobility and F-formations. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST 2012)*. Cambridge, USA. https://doi.org/10.1145/2380116.2380121

[23] Jakob Nielsen. 1989. Coordinating User Interfaces for Consistency. *SIGCHI Bull.* 20, 3 (January 1989), 63–65. https://doi.org/10.1145/67900.67910

[24] Roman Rädle, Hans-Christian Jetter, Nicolai Marquardt, Harald Reiterer, and Yvonne Rogers. 2014. HuddleLamp: Spatially-Aware Mobile Displays for Ad-hoc Around-the-Table Collaboration. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces (ITS 2014)*. Dresden, Germany, 45–54. https://doi.org/10.1145/2669485.2669500

[25] Jun Rekimoto. 1997. Pick-and-drop: A Direct Manipulation Technique for Multiple Computer Environments. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST 1997)*. 31–39. https://doi.org/10.1145/263407.263505

[26] Jun Rekimoto and Masanori Saitoh. 1999. Augmented Surfaces: A Spatially Continuous Work Space for Hybrid Computing Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 1999)*. Pittsburgh, USA, 378–385. https://doi.org/10.1145/302979.303113

[27] Iván Sánchez, Jukka Riekki, Jarkko Rousu, and Susanna Pirttikangas. 2008. Touch & Share: RFID Based Ubiquitous File Containers. In *Proceedings of the 7th International Conference on Mobile and Ubiquitous Multimedia (MUM 2008)*. Umeå, Sweden, 57–63. https://doi.org/10.1145/1543137.1543148

[28] Dominik Schmidt, Julian Seifert, Enrico Rukzio, and Hans Gellersen. 2012. A Cross-device Interaction Style for Mobiles and Surfaces. In *Proceedings of the Designing Interactive Systems Conference (DIS 2012)*. Newcastle Upon Tyne, UK, 318–327. https://doi.org/10.1145/2317956.2318005

[29] Arne Schmitz, Ming Li, Volker Schönefeld, and Leif Kobbelt. 2010. Ad-hoc Multi-Displays for Mobile Interactive Applications. In *Proceedings of the 31st Annual Conference of the European Association for Computer Graphics (Eurographics 2010)*, Vol. 29. 8.

[30] Mario Schreiner, Roman Rädle, Hans-Christian Jetter, and Harald Reiterer. 2015. Connichiwa: A Framework for Cross-Device Web Applications. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA 2015)*. Seoul, Republic of Korea. https://doi.org/10.1145/2702613.2732909

[31] Khoovirajsingh Seewoonauth, Enrico Rukzio, Robert Hardy, and Paul Holleis. 2009. Touch & Connect and Touch & Select: Interacting with a Computer by Touching It with a Mobile Phone. In *Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI 2009)*. Bonn, Germany, 36:1–36:9. https://doi.org/10.1145/1613858.1613905

[32] Khoovirajsingh Seewoonauth, Enrico Rukzio, Robert Hardy, and Paul Holleis. 2009. Two NFC Interaction Techniques for Quickly Exchanging Pictures Between a Mobile Phone and a Computer. In *Proceedings of the 11th International*

Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI 2009). Bonn, Germany, 39:1–39:4. https://doi.org/10.1145/1613858.1613909

[33] Manas Tungare, Pardha S. Pyla, Pradyut Bafna, Vladimir Glina, Wenjie Zheng, Xiaoyan Yu, Umut Balli, and Steven Harrison. 2006. Embodied Data Objects: Tangible Interfaces to Information Appliances. In *Proceedings of the 44th Annual Southeast Regional Conference (ACM-SE 44)*. Melbourne, USA, 359–364. https://doi.org/10.1145/1185448.1185529

[34] Brygg Ullmer and Hiroshi Ishii. 2000. Emerging frameworks for tangible user interfaces. *IBM Systems Journal* 39, 3.4 (2000), 915–931. https://doi.org/10.1147/sj.393.0915

[35] Daniel Wigdor, Hao Jiang, Clifton Forlines, Michelle Borkin, and Chia Shen. 2009. WeSpace: The Design Development and Deployment of a Walk-up and Share Multi-surface Visual Collaboration System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2009)*. Boston, USA. https://doi.org/1518701.1518886

[36] Andrew D. Wilson and Raman Sarin. 2007. BlueTable: Connecting Wireless Mobile Devices on Interactive Surfaces Using Vision-based Handshaking. In *Proceedings of Graphics Interface, Conference on Graphics, Visualization and HCI (GI 2007)*. Montreal, Canada. https://doi.org/10.1145/1268517.1268539

[37] Christian Winkler, Markus Löchtefeld, David Dobbelstein, Antonio Krüger, and Enrico Rukzio. 2014. SurfacePhone: A Mobile Projection Device for Single- and Multiuser Everywhere Tabletop Interaction. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems (CHI 2014)*. Toronto, Canada, 3513–3522. https://doi.org/10.1145/2556288.2557075