emergence from the laboratory to the home of many of the ideas and innovations investigated in the field of telemedicine and telecare is the lack of a cheap and reliable instrument for performing numerous and varied field investigations particularly in the home.

The home is a very hostile and unpredictable environment in comparison to the laboratory and it is difficult to manage successful deployment of standard PC based solutions because they are easily accessible and vulnerable to modification and re-configuration. The standard PC and laptop with well known operating systems invites interference by both expert and novice.

We are unaware of any suitable off-the-shelf hardware and software available in the market that could be used to support such research needs that would not require extensive modification and customisation. Therefore, this paper describes the development of a 'gateway' device, which may be used as a hub to download data from medical monitoring devices, environmental sensors, gather information from psychological surveys and schedule physiological tests, with the ability to monitor the device remotely. This paper considers game consoles (such as the Sony PlayStation 2 and 3, Microsoft Xbox and Microsoft XBox 360) as a hardware platform for the gateway device. We use the term ‘gateway device’ to indicate the use of a dedicated computer that is placed in a person’s home, place of work or even a moving vehicle and is used to coordinate a number of other digital and electronic devices that are connected to it. These gateway devices, provide the basis for a flexible biomedical data acquisition and management technology that can be deployed to all environments.

The programming language Java with an OSGi framework was chosen as our software platform because of its suitability for residential gateway devices and relative portability and security ([3]; [4]; [5]). Much of our previous work and our future investigations are focusing on the popular notion of the smart home and [6] [7] and there are many reasons for developing a cheap, reliable and resilient hardware and software combinations that can be deployed to the field and in particular the home environment for which the Java language and OSGi framework appear most suitable.

The first half of this paper will proceed to describe investigations undertaken in order to identify and satisfy the base hardware requirements with two game consoles, (i) the Sony PlayStation-2 (Sony Computer Entertainment, PLAYSTATION 2, Australia) and (ii) Sony PlayStation-3 (Sony Computer Entertainment, PLAYSTATION 3, Australia) and the relative strengths and weaknesses of each console. The second half of the paper will outline investigations into the software requirements and how the OSGi framework supports or with modifications can support these requirements, including an outline of our implementation.

II. REQUIREMENTS

The SmartData project aims to collect data with relatively simple, off-the-shelf electronic devices that monitor heart rate, physical activity, brain activity, temperature etc., and examine this in the context of self- reported results from participant's response to psychological surveys (such anxiety, emotion, mood etc.) and certain physiological tests (reaction time, EEG etc.). However, the data from such devices must be constantly downloaded onto a desktop computer, and the effort required to manually collate and analyse this data using conventional methods such as paper-based forms makes such research substantially time-consuming and expensive. Furthermore, once a research program has been started, it can be very difficult to change aspects of it during its progression, especially if the participants are geographically displaced from the main research site for long periods of time, as would be in the case of truck drivers during long road trips.

It is apparent that a device is required that could be placed in a participant's home, work or vehicle to act as a hub or gateway to collect data and feed it back to the researchers in a central location. This “gateway device” must be capable of gathering data non-intrusively and allow participants to provide response to electronic surveys to support psychological profiling. In order to monitor participants’ progress without physical access to the gateway device for long periods (sometimes up to a few weeks), the device would need to be controlled remotely, able to download updates, as well as subsequently upload research data after the research program commences.

It was also a requirement that the device should be readily available and easily obtainable. Even better if the device was a commodity while not an immediate or interesting target for hackers or denial of service attacks. It has been noted in the literature as devices for collecting important physiological and psychological data become more prevalent and critical to quality of life then the consequences of connectivity to the Internet and the inherent dangers of hacking and denial of service increase [8]. It was also a requirement that the device was able to work with minima of support equipment, be self contained and able to work with a standard TV: on the basis that all homes are likely to have a TV set. The requirements support the ambition of a device that can be readily deployed worldwide to all field situations, so no legitimate scientific investigation is hampered by lack of electronic resources and does not place a burden on potential participants and scientific endeavor to source additional monitors or anything but the standard power supply of the locality.

The use of a PC as a solution was ruled out because it is too hard to control the operating system versioning, software updates and numerous other malicious pieces of software which can find their way on to PC once connected to the Internet. As it was a requirement to connect to the Internet standard operating systems like Microsoft XP or Vista are too hard to maintain in a know configuration. Often Microsoft itself will install updates and changes in the background. This feature can be disabled but it is easily enabled too. It is also too easy for participants and others to interfere with these common operating systems. A piece of instrumentation for scientific investigation needs to be as secure as possible and in a known state of configuration to ensure reliable and repeatable measurement.

It is therefore logical to look for a more resilient operating system and a less well known platform that is less inviting and therefore through obscurity and lack of interest protected from many of the configuration management issues encountered with PC based hardware.

A. Hardware

The above requirements suggest a basic architecture of the proposed system, which would be similar to that in Figure 1.

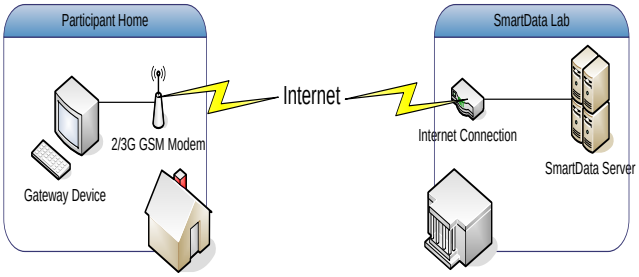

Fig. 1. Architecture of SmartData system

The type of software required to collect the data from such external devices and to allow updates to program schedules, is specialized and built on top of an OSGi based framework, and was developed by the SmartData project to support it requirements: in awareness that many of these requirements were generic to support field based investigations and future research in the field. We wanted to reduce the time needed for software testing during development by requiring that the target device support a full Java Standard Edition runtime environment and not have the software satisfy stringent performance requirements. This means that the device cannot be a resource-constrained system and should be capable of running a 32-bit, pre-emptive, multitasking operating system with virtual memory management and memory protection. The device had to be cheap to be deployed in large numbers but customisable to permit the SmartData software to run. There was a strong desire for the device to be able to connect to participant's television sets to reduce the costs associated with purchasing separate display units. A simple way to connect the monitoring devices using a common peripheral technology such as Universal Serial Bus (USB) is required: many of the commercially available sensors and activity monitor devices need to be worn or carried by the participant whilst they are not at home, and then connected to the gateway at a later time for data transfer to a remote server One example of such as an activity monitoring device is the Actigraph GT1M Monitor, which collects and records body movements [9]. Large data sets would be derived so the device needs secondary storage. It is also important to be able to lock down the device so that participants could not misuse the device or interfere with its normal operation.

B. Software Platform

As this box will be deployed “out in the field” and has to be kept running for long periods, a software platform with high reliability that could be updated whilst the system is still running is required. Software with relative portability was also desirable in order to deploy the same or similar software on later gateway devices to be modified for deployment in trucks as well as to collect ‘real field’ driving data. We wanted flexibility to change hardware platforms and operating systems where we deemed it to be necessary. Java was chosen over languages such as C/C++ because it tends to be more portable, provides fine granularity of in-process security through the Java Permissions Application Programming Interface (API), and it's lack of direct pointer-access can help reduce common programming errors that are fatal to program execution (such as null pointer exceptions and dangling pointer dereferences) and cause security issues such as buffer overflow exploits [10].

Java also served as a base for the OSGi platform, which was designed to run on residential gateway devices that have continuous uptime and need to be managed and updated remotely, without user intervention. It extends Java to provide a structure that can be used to build inter-changeable software components that are loosely coupled in the form of services, allow fine-grained security configurations, and can be updated without restarting the framework [11]. There are multiple vendors that implement the specification ([12]; [13]; [14]; [15]). This gives more flexibility in choosing an environment that suited our research needs and makes it easier to switch implementations later if required.

Java was also a convenient language and run time environment already supported as a deployment and server environment for wireless sensor motes (namely the TMote Sky and TMote Invent [16]). The project intended to use these motes to provide wireless sensors capable of performing a number of tests, such as reaction time assessment tests.

III. Selecting Appropriate Hardware

We examined multiple computer-type devices suitable for our needs. We first looked at an emerging field of devices used such as home gateways which are Personal Video Recorders (PVRs) for watching and recording broadcasted television (TV) program and Set Top Units for watching digital TV and subscription TV services. They are mass-produced and built to interact with a TV, which makes them suitable for our application. Unfortunately, many are tied to proprietary platforms without much scope for modification, such as set top units distributed with cable-TV services, and others are not on the market for long enough so that we can make a firm decision.

PC-based systems were also considered as they are considerably cheap and have good software and peripheral support due to widespread use. Apart from being able to support a greater range of operating systems (such as Windows or Linux, etc.) there is also good support for development due to their widespread use. It has excellent backwards compatibility, and there is a wide variety of vendors that produce PC-based hardware, making it very difficult to be “locked-in” to one manufacturer. It is also possible to source hardware boards that have most peripherals integrated.

The main concern with PCs was how easily they could be used by the participants to install another operating system, whereas dedicated devices tend to require expert technical skills in order to modify them. There is also considerable variance amongst all the PC-based products in the market with most products having short lifecycles before a new variant is introduced, making it harder to predict performance if the hardware needs to be frequently changed. Using newer hardware means that we would need to thoroughly test new software drivers, which in our experience are more unstable in newer versions. This would be a more practical option in the future if we could select products with long lifecycles, and a

simple method could be identified to “lock-down” the devices.

The last option being considered is game consoles. Similar to PCs, they are relatively cheap due to mass production and remain compatible with their software over the product lifecycle. Their product lifecycles are quite long, for example, the PlayStation One was available in Japan in December 1994 [17] and discontinued from March 2006 [18]. The PlayStation 2 was introduced in March 2000 [17] and continues to be available, and the Microsoft Xbox was introduced in March 2000 [19] and discontinued in 2006. They also connect easily to TV sets. Customisation is a big concern as most consoles are proprietary and like set top units are not easily modifiable. There is a risk of vendor “lock-in” because only one company or organization typically manufactures each model. The only commercially available consoles identified to allow customisation without purchasing prohibitively expensive Software Development Kits were the Sony PlayStation-2 and PlayStation-3, both of which can run Linux with fairly inexpensive modifications supported by the manufacturer. The Xbox (Microsoft Xbox) was also a consideration as it has been shown to run Linux, but it required a legally dubious and unsupported “mod-chip” in order to boot Linux and changes to its controller ports in order to access its USB hub. We ruled this option out because we wanted minimal hardware changes and less uncertain legal issues.

A. PlayStation 2

The PlayStation-2 (PS2) features multi-core MIPS architecture called the “Emotion Engine” (clocked at about 290MHz) with cores designed for handling image processing, video-decompression and vector calculation [20]. There are separate processors for handling graphics, audio and PlayStation-1 games. It has about 32MB of RAM, composite TV-OUT or RF-OUT, two controller ports with memory card slots, and two USB 1.1 ports at the front.

Because of these powerful hardware features, many technically advanced purchasers of the PS2 expressed an interest in running Linux on their consoles. Due to the copyright protection implemented by Sony to prevent pirated games from being executed in the console, it was not possible to simply create a bootable Linux disc, plug in a hard drive, and install Linux. The cost of licensing an official development kit for the PS2 is also prohibitive enough so that only commercial game and application developers can afford to purchase one.

Sony published a “PlayStation-2 Linux Kit” (Sony Computer Entertainment, Sony PlayStation 2 Linux Kit) that was cheap enough for purchase by consumers. The kit included a hard disk, mouse and keyboard, a disc for booting GNU/Linux, another disc containing a Red-Hat based GNU/Linux distribution, a computer monitor adaptor and an expansion card containing a network card and hard disk connector for attaching the hard disk. The kit requires older hardware versions of the PS2 that contains an expansion slot for plugging in the add-on card (specifically models before the SCPH-7000), which is something that does not appear on the newer 'slim' hardware and makes it impossible to use the PS2 Linux kit [21] with newer hardware.

This kit contains versions of Linux and GNU software that is outdated by today's standards (most sourced from the time the kit was developed), running the Linux kernel 2.2 and a GNU Compiler Collection (GCC) 2.95. This made it more difficult to cross-compile and run newer software and more difficult to use some of the newer hardware devices (such as the Actigraph) that we intended to use. As the PlayStation-2 is a specialised hardware platform with a unique variant of MIPS architecture and peripherals, the official Linux and GNU maintainers do not provide support.

Most of the software required for the research was compiled directly on the PS2 hardware or by using a cross-compiler that was obtained from the PlayStation-2 Linux Community website [21].

The PS2's CPU architecture lacks two useful instructions, 'll' and 'sc', which are used together to implement atomic swap-and-load instructions for simple synchronisation primitives. These instructions are used extensively in some open source Java virtual machines for synchronisation between threads instead of kernel based locks. For synchronisation that only needs to last for a few CPU cycles (such as incrementing an integer value), this type of locking is more efficient as it avoids the overhead of a system call and context switch into supervisor mode. It was determined not worthwhile to modify the virtual machines and adapt them for system-call based locking because these user-space locks were found to be used extensively in their source code. Furthermore, it would seriously degrade performance by adding many more system calls for simple synchronisation needs. The Kaffe open-source Java virtual machine [22] was the only one found to have been specifically made compatible for the PS2, whilst others such as JamVM and cacaovm, still relied on these specific locking instructions found in normal MIPS architectures.

Another major problem identified was with software performance. Most of the Java virtual machines trialled did not support Just-In-Time compilation for the MIPS architecture, or if they did (in the case of Kaffe), they did not have the support for the PlayStation-2's unique architecture. This meant we had to turn on the slower, C-based interpreter for all the virtual machines trialled. This meant that the performance was unacceptably slow and unable to even start up the OSGi framework due to the small amount of RAM and slow execution speed.

Assessing other virtual machines such as JamVM yielded mixed results. The compiler tool-chain being used was too old (GCC 3.x) to support newer code. We did consider the option of modifying the source code of these programs so that they could be compiled with the older GCC however this was beyond the team’s level of expertise.

One of our more significant reasons for not continuing with the PS2 was the lack of available hardware. The Linux Kit is only available for some regions now (having sold out in the United States and other places) and no longer includes the add-on card needed for network access and to plug in the hard disk; it only has the DVDs and VGA cable. Because it requires the older, larger PS2 and the discontinued expansion port, both these parts need to be carefully located and purchased second hand. This means that there is no guarantee that those parts will be readily available to carry out the proposed research. Even if the slim PS2 models with the cut-down Linux Kit could be

used, there is no room or place to connect and mount a hard disk inside the unit.

The main support for the kit was obtained from the PlayStation-2 Linux Community Web Site [21]. Amateur software developers managed to port the PlayStation-2 specific changes to a GCC 3.3 and a Linux 2.4 kernel and published their results on this website, but due to a lack of developer resources, they have not been able to port the patch sets to more recent GCC and kernel versions (Linux 2.6 and GCC 4.x). Additional updates to the kit from Sony for newer software versions on the PS2 could not be found.

The official development kit was also not a viable option, not only because of the cost alone, but also because an agreement would need to be executed with Sony who would have to approve the software we wanted to produce and force us to print our own discs. The costs involved with the option were considered too impractical for our project and so we decided not to pursue it.

B. PlayStation 3

The PlayStation 3 (PS3) has a multi-core PowerPC based Cell architecture designed for intensive multiprogramming and graphics/vector calculation. It was built for complex three dimensional computer games that need to perform intensive physics and graphics calculations. It also has 256MB of general purpose RAM and a further 256MB of graphics RAM. Unlike the PS2, the PlayStation-3 comes with a built-in hard disk drive, Bluetooth technology, wireless controllers, USB 2.0 ports and 802.11b/g Wi-Fi Wireless Networking adaptor [23].

The graphics output on the PS3 is also more flexible. The PlayStation-2 required a PAL (Phase Alternating Line) or NTSC (National Television System Committee) compatible TV set or it was difficult to find computer monitors that supported required "sync-on-green" [20], but the PS3 has HDMI (High Definition Multimedia Interface) and composite outputs. This allowed the connection of a digital computer monitor via an HDMI-to-DVI adaptor as well as a High-Definition or normal analogue television (PAL and NTSC).

GNU/Linux and other operating systems are supported natively on the PlayStation-3 via the "OtherOS" facility in the System Dashboard. This is set up by loading a bootloader image file from a CD, DVD or USB flash drive with the PS3 Dashboard, which copies it into internal flash memory and uses it to boot-strap the other operating system.

Similar to the PS2 RTE, Sony has implemented protection of the graphics hardware through a "hypervisor", which sits between the PlayStation-3 hardware and the OtherOS. It still allows access to the processor cores and main RAM, but only provides a framebuffer for graphics output, not full graphics acceleration as it does for games [23].

Running Java virtual machines on the PlayStation-3 has been much more successful. IBM produces a pre-compiled version of a Java 1.5 and 1.6 virtual machine for its PowerPC architecture that runs directly on the PS3 [24], and JamVM is also available. We had no problems using them to start up an OSGi environment with the Knopflerfish and Equinox OSGi implementations ([13]; [14]).

We were able to download and install most of the software we required using pre-compiled packages for the distribution that we were using (most packages came from the PowerPC variant of each distribution). The PS3's Cell chip had enough power to compile and run the Java VM and full-profile OSGi framework too. Each distribution run a newer GCC 4.x variant so there were no significant compilation issues.

The above makes the PlayStation-3 a more attractive option, and it appears that Sony supports continued work on the Linux kernel and some user-space utilities for running Linux on the PS3 [23]. At this point in time, it is possible to compile unmodified versions of the latest Linux 2.6 kernel for the PlayStation-3. However, it isn't clear how long Sony will continue to support Linux on the PS3, and they could easily withdraw their support in the future. If Sony decided to do this, we would have to reconsider the other hardware platforms for our gateway device, such as PCs.

Initially, the PS3 was not a compelling option as it is more expensive than the PS2. Even after pricing the extra hardware and software that needed to be purchased for the PlayStation-2 to run Linux, the PS3 is still more expensive compared to the PS2. The PS3 is not expected to become much cheaper for some years. However, given the relative difficulty of finding the necessary PS2 hardware, compared with immediate retail availability of a PS3 system, the extra cost may be justified.

IV. HAL AND OSGI FOR DEVICE DETECTION

There is a need to collect data about research participants using devices that are plugged into the gateway. The gateway collects the data from the devices and stores it until it can be uploaded to a laboratory server. Because this needs to be done automatically (i.e. without the intervention of the participant or the researcher), the system needs to be able to recognise the devices when they are plugged in, load the drivers for them and download and/or reconfigure them at the same time. Any functions that need to be performed using the device will need to be activated as well. This device management requirement is a consistent issue for software designed to support autonomous systems development and derives from the requirement, that devices are self-configuring and self-managing, but more often than not, current off-the-shelf software platform cannot or do not support this requirement well. Device drivers frequently need to be selected by the user and installed manually. Making matters more difficult, users need instruction in how to use the provided software in order to manipulate such devices.

In the case of the gateway device to support our field research this major issue of device management was a focus for our investigations. The aim was to utilise the chosen hardware platform, the PS3, and augment it through software to satisfy the requirements outlined earlier. The means to automate device management were investigated which involved combining the capabilities available from an implementation of the Hardware Abstraction Layer (HAL) and an implementation of the OSGi specification. The HAL provides rich device information and events required to detect and distinguish between devices and the OSGi provides an application platform that permits devices to exist as dynamic services, which can come and go at any time and automatically execute the relevant application software. The integration of these two components would create a software device management capability for dynamic device discovery and automatic driver

connection, upon which we could build applications that automatically retrieve the physiological, psychological and environmental data and send it back to researchers in the laboratory.

The initial research investigation was undertaken to learn how an implementation of this concept in the form of device driver(s) adapted as an OSGi services might be realised. We then provide as an example purpose-built driver for a data collection device as a demonstration of the utility of the integrated software platform and how it may be deployed to a hardware platform such as the PS3.

V. Establishing A Specific Example Device To Investigate Device Management

Existing systems typically detect devices, load a driver written for the device, and then expect the user to manipulate the device themselves. For example, an activity monitor device such as the Actigraph [9] a personal electronic device that records the "activity" of a person who is wearing it. It is usually necessary for the users to plug it in and wait for the device to be detected. After that, the users have to activate a program that is designed to download the data and send it back to a remote site. This requirement that users are responsible for knowing how to manipulate the device and remembering to activate a specific process, both of these are unsuitable for unobtrusive field based research. We do not want to have to rely on participants remembering to upload their data, and knowing how to do so. It would require the researcher to train participants on how to use each device within the scope of the research and limit easy deployment to the field of updated or new sensor devices as research progresses. Furthermore, such an approach requires that the system has the drivers pre-installed before deployment. It is unacceptable to require that the participants know how to install the drivers or even how to activate the install process. An autonomous field based system for our research cannot expect or rely on research participants to do anything other than plug the Actigraph into the gateway.

Something more intelligent than the current driver models is needed. The drivers on our chosen operating system (GNU/Linux) classify devices into generic groups such as "storage" or "serial" or "display, and then exposes them in groups to the application programs. The operating system makes no hard attempts to distinguish them based on their more individual capabilities in its user-level interfaces to application software.

For example, another useful sensor device used by the project TMote like the Actigraph devices appear as "serial" devices to the operating system, along with any serial ports on the back of the system. It is worth noting before continuing that the TMote device is a small electronic device that can be customised with an operating system to control its own packet radio device and monitor its sensors and button inputs. They are used to form "mesh networks" of devices that communicate between each other by relaying short messages over their radio interface. They can be connected to our gateway through a USB connection.

Each of these will be exposed as serial port devices to the application programs. The problem for an application program which may only know how to communicate with one type of device (e.g. an Actigraph downloader), is how to pick the right device from those connected to the system as it's unable to differentiate between devices that appear as "serial" devices.

Operating system developers have tried to address this problem by providing extra information (a.k.a. meta-data) about devices to application developers. On Linux, this is a combination of virtual files on the /proc and the /sys file systems. This information can be difficult to parse, is often in different formats and can be subject to change between kernel versions.

The requirements for field based autonomous gateway continue to change as well. There is ongoing research into the best methods by which to conduct tests. As the research continues to trial new methods, we need to be able to easily adapt the system to new external peripherals: sensor devices usually connected either using USB or Bluetooth connectivity. We may need to do this even while the gateway is deployed. Our driver model needs to allow the addition of new devices so that we can utilise them immediately, without stopping the running gateway.

A. Software Requirements for Device Management

The relevant software requirements extracted from the considerations outlined above are listed as follows:

- the gateway shall be able to detect when external devices are attached to the system
- shall be able to detect when external devices are removed from the system
- shall attach the Drivers and software from a device when it is attached to the system
- shall detach Drivers and software from a device when it is detached from the system
- shall update any status display associated with external devices when they are attached and detached from the system.

There are also some other requirements that are useful to include in the scope of this investigation to the support the broader needs of autonomous data collection in the field:

- the gateway shall be able to download data from External Data Collection Devices when they are directly plugged into the gateway
- shall be able to reconfigure External Data Collection Devices that support it
- shall be able to upload new firmware or software onto External Data Collection Devices that support it
- shall be able to format and configure the secondary storage of External Data Collection Devices that support it.

The capabilities of each device will be exposed on a device by device basis; device capabilities will not necessarily be made generic across a class of devices unless it is desirable to do so e.g. it may be reasonable to have a generic process for recording test results, but not use a common file format.

The requirements as outlined suggest the need for a highly dynamic software environment, where drivers and client software for devices can be loaded when they are connected, and appropriately unloaded when they are detached again. Devices need to be discovered automatically using appropriate

meta-data to match the right drivers to the right devices. Because research participants may not know much about the devices we give them beyond what they do and how to plug them in, the system needs to use this meta-data to locate and automatically run the right programs for each device.

Finally, so that we can introduce new devices to the system whilst it is deployed and update the software for those that are already installed, we need to be able to add and replace drivers from the system whilst it is running.

1) Software Platform Framework Considerations

OSGi [11] was selected as a programming model within the Java environment as it supported our requirements for dynamic service provision. It is a Java specification for a dynamic environment that allows loading and unloading of software components (known as "bundles") as the Java virtual machine is running. It separates components from each other in such a way that promotes the use of carefully designed interfaces that hide the implementation from the caller (decoupling).

This separation is achieved through the use of "import" and "export" packages. The import packages are a list of Java packages that the bundle requires to be provided by another bundle. Export packages are a list of packages that a bundle wants to make available to other bundles. A bundle can only import packages that have been exported from another bundle otherwise it remains "unresolved" and cannot start. Consequently, a bundle can keep some of its code private to other bundles, and similarly, a bundle may choose not to import a package and hence won't be able to access it. This achieves greater modularity and more explicit separation of code through "information hiding".

Bundles are "wired" together based on a pre-specified set of dependencies, and communicate through services that may be registered and consumed by a bundle when it is running. Services may come and go at any time, they may consume other services (forming service dependencies) and they may expose meta-data about themselves as properties. A service is just a Java object, published into the OSGi service registry under an exported interface. A service may also publish a list of "properties" that describe something about this particular service. They are vital for distinguishing between different services published under the same interface e.g. two serial port services, COM1 and COM2, might be published under the SerialPortDevice interface, but a property called "name" would be used to distinguish them.

Using OSGi, we were able to program against a model that allowed components to be upgraded, removed and added from/to the system whilst it is still running. As components may have dependencies on each other, the framework provides the means for them to adapt to changes in their dependencies by degrading or improving on their services. This means that it's possible to update individual components whilst the framework is in operation, only requiring that the dependant services degrade their operation whilst the update is in progress.

This dynamic model forces us to program in such a way that our components were adaptable against the changes in the system. We are able to write new software components, such as drivers for a new medical research device such as the Actigraph, and install them into the remotely deployed system without having to restart it or shut it down. In combination with a remote management system, this is desirable for autonomous field deployment as it saves precious time and money in not having to send equipment back to the laboratory to be updated. This allows already deployed devices to be quickly updated or corrected whilst in the field.

The OSGi standard also provides optional sub-specifications for many services that are useful to our style of application, including configuration management, deployment management, per-component security and device management. It also has more than one implementation available ([12]; [13], [14]; [15]), with a combination of open-source and commercial providers implementing some or all of the core and optional sub-specifications. Many of the components can be mixed and matched across framework implementations, which gave us maximum flexibility in how we configure our deployment platform.

2) Using OSGi Device Access

OSGi contains a specification for device management called the "Device Access Specification". It defines a generic driver/device model so that devices and drivers may be exposed as OSGi services, and automatically linked together based on a numerical matching algorithm. It deliberately doesn't specify a device model so that different device models can be supported. This model works well for devices, because as devices can come and go, the services that represent them can be added and removed too. As programmatic services, their functionality can be exposed through a software interface to other components in the framework. An implementation of the specification provides an invisible helper service called the "Device Manager" that provides the driver to device refinement algorithm and process. When we refer to the Device Access Specification, we are referring to the specification and a corresponding implementation, such as Eclipse Equinox's org.eclipse.equinox.device [13] bundle.

3) Hardware Abstract Layer

Hardware Abstraction Layer (HAL) [25] is a software daemon that runs on GNU/Linux and similar UNIX-like systems to provide meta-data about the devices connected to the system and events for device attachment and detachment. It aims to simplify the discovery of devices by software programs. It can classify devices by their capabilities as well as adding extra meta-data about each device, such as its manufacturer and model. For example, if a device is a serial device (such as a serial port or a USB Serial adapter), it will be exposed with a capability of "serial", regardless of how or where it is connected, but it is also possible to distinguish it further, if it is an RS232 port or an Actigraph. HAL provides a device model for classifying devices, well-defined properties for getting meta-data about devices, and a set of standard events for different classes of devices. It additionally provides extra functionality for manipulating some classes of devices (e.g. mounting file-systems).

4) Capabilities and Limits

a) OSGi Device Access Specification

The Device Access Specification defines a limited functionality for automatically wiring device services to driver services through a matching algorithm. The default matching algorithm works by asking each driver to test how well it can refine a particular device. This algorithm is replaceable with the use of a Driver Selector service, but it cannot be specified on a per-driver or per-device basis (only for the whole system).

Each driver implements an interface that allows the Device Manager to test device services for their suitability (the "matching algorithm") and to attach them to a selected device service. They also provide a way for drivers to "refer" the Device Manager to another driver that the driver considers more suitable.

The driver and device services that are running in a framework can be arranged in any number of topologies. The OSGi Device Access Specification defines the following styles of drivers that can be developed with its devices model ([26]; [27]; [28]):

- **Base Driver** – a driver that only exposes new device services to the framework
- **Refining Driver** – a driver that consumes a device service, refines its functionality and exposes a higher-level device service
- **Pure Consuming Driver** – a driver that consumes the functionality of a device service, but does not expose any devices
- **Network Driver** – a driver that optionally consumes a device, and exposes one or more device services representing devices on a network
- **Composite Driver** – a driver that exposes the different parts of a complex device as separate device services
- **Referring Driver** – a driver that participates in the refining process, but hands out the ID of another driver that should be used instead.
- **Bridging Driver** – a driver that exposes a device service from one category but refines a device service from another category
- **Multiplexing Driver** – a driver that consumes a number of device services but only refines it to one aggregated device service

The specification also provides for Driver Locator services, which are specially written services that can download or locate the driver for a device when given the device that needs to be refined

5) OSGi Framework Implementation

An important infrastructure component is the OSGi framework implementation. As the software for the gateway device was being developed under the Eclipse Equinox framework ([13]; [15]). The base part of Eclipse Equinox provides the key functionality needed for an OSGi environment as described earlier.

a) Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) [25] uses the DBus [29] inter-process communication system to publish a list of "device objects" that correspond to each device in the system. It usually gets its meta-data from a combination of system devices, kernel device information (such as the /proc filesystem) and specially written device information files which get merged into the device information tree.

Because of the DBus session bus that HAL uses, each device object behaves like an object in the object-oriented sense i.e. they have properties that can be queried and changed, and there are methods that perform a function. HAL's device objects contain a rich set of properties that can be used to determine their capabilities, get human-readable descriptions and work out where they fit into the device tree. It also provides some specific refinement classes for certain types of devices e.g. volume devices can be manipulated via a "Mount" and "Unmount" method.

DBus also provides for events from objects. HAL publishes a number of useful events, not limited to:

- Attachment of a device from the system
- Detachment of a device from the system
- Lock/Unlock of device
- Device properties change

These are the types of events needed to support the core requirements around device attachment and detachment. Essentially the investigation planned to bridge these events and the meta-data from HAL into the OSGi framework, using each of these device objects as an OSGi device service. The rich property set provided by HAL can be mirrored into the OSGi service registry through service properties as well as through accessor methods on the device interface itself.

B. Example Driver

In order to demonstrate the usefulness of the approach and the utility of the developed software the investigation planned to use the Actigraph "activity monitoring device" as a representative sensor device. It has a configurable sample rate and step counter, and uses a USB connection to download data and upload a configuration. The USB communications utilise a USB Universal Synchronous/Asynchronous Receiver/Transmitter (USART) chip, which exposes the device as a virtual serial port to the computer, given the right drivers are loaded into the operating system kernel.

This required the development or extension of three drivers: a base driver for locating the devices in the system, a serial port driver that detected serial port devices and exposed a serial interface, and an Actigraph refining driver that uses a serial interface for setting up and downloading data from an Actigraph.

The base driver was implemented using a third-party dbus-java component to communicate with a DBus daemon, upon which HAL would be connected. The Java serial port driver was implemented out of the RxTx [30] component which provides serial port access to Java programs on many Unix platforms (including Linux).

For the Actigraph, the driver needed to be implemented in

Java from scratch. Unfortunately, there was no information available on the protocol that the device used to communicate with the computer, so it had to be reverse-engineer. This was done using a serial port monitor program whilst running the device's coupled software, ActiLife [9].

It is worth mentioning that an additional family or network of sensor technology was also investigated and constructed from TMote device for which an initial build of a refining driver was considered. TMotes are wireless sensor network devices which we were using for another type of physiological testing. They are USB serial based like the Actigraph. We needed to be able to detect when they were plugged in so that we could launch an associated program, but we discovered that we could perform this operation through a special script addition to the HAL daemon.

1) Architecture of the OSGi-HAL

As can be seen in Figure 2, HAL mounts itself onto the D-BUS daemon in order to provide access to device information and signals when devices are attached and detached. Using Java D-BUS, it is possible to connect into the D-BUS daemon and provide Device types using the OSGi Device Access Specification in a set of generic OSGi Base Drivers. These Base Drivers expose Device interfaces that OSGi will attempt to further refine, and from here can be built Refining Drivers that expose a refined interface on a Device, or Consuming Drivers which can be used by a software application to manipulate a device.

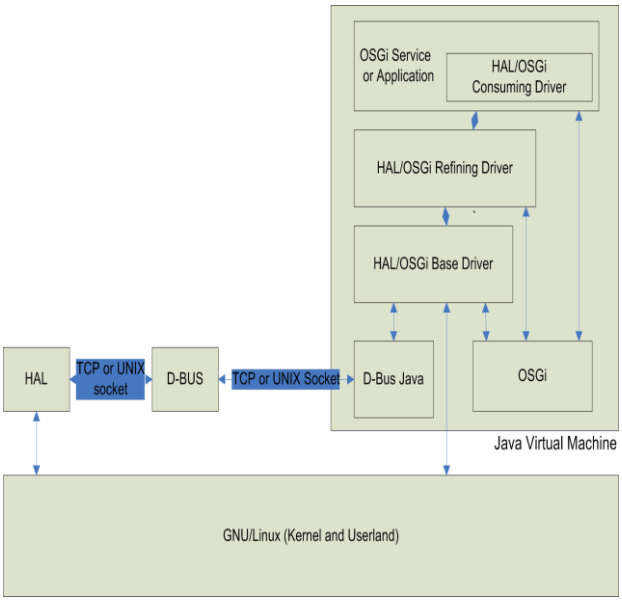

Fig. 2. Architecture of OSGi and HAL components

For example, to build a Base Driver for connecting to serial port devices that gets its information from HAL. It exposes a Device interface that can be consumed as a raw serial port, or if OSGi can find a Refining Driver that matches the device attached to this serial port (through the match() method on the Driver interface), it will load that driver instead. A serial driver is placed above the refining driver for the Actigraph device, and finally a consuming driver that uses the Actigraph driver to download data and send it off to the remote server in the laboratory.

C. Services Design (Runtime)

The OSGi Device Access Specification provides a method for graphically describing the wiring between driver and device services. This method and notation is used in the diagram below in Figure 3 to show the different driver and device classes and how they are to be wired together at runtime in the case of the Actigraph.

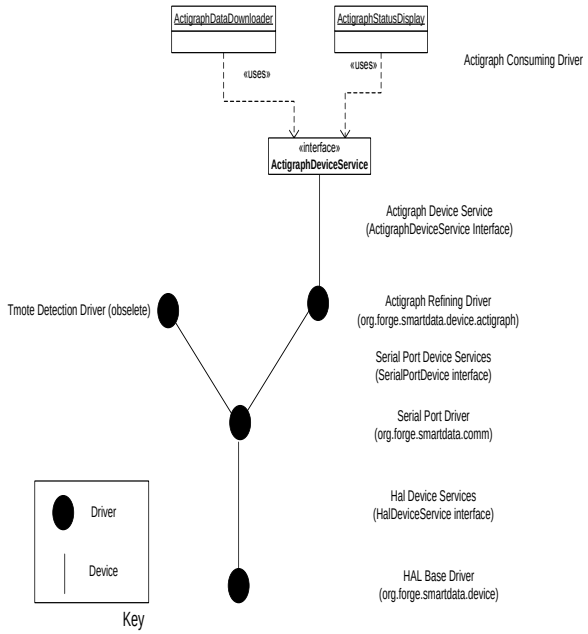

Fig. 3. Driver Device Wiring

Each driver and device represented by a node in the diagram above represents an OSGi service. When we publish a service into the service registry, we use an interface for an instance of that service. More than one service may be published under the same interface, so they are distinguished through the use of OSGi service properties, which are name-value pairs (usually string types), attached to the service instance.

The wiring between drivers and devices shown above is known as device refinement, because the driver uses the functionality of the lower device service to expose a higher-order device that refines its capabilities. This wiring takes place via a matching algorithm in the Device Manager.

1) Automatic Driver and Device Matching

OSGi's Device Manager determines that a device is a device service if it is published under an interface that is derived from the OSGi Device interface and/or contains a DRIVER_CATEGORY property. It detects drivers if they are published under the OSGi Driver interface and distinguishes them by their DRIVER_ID field, Figure 4.

2) Expandability

The current design provides for full enumeration of all the devices in the system, which means that it should meet the criteria for expandability to support new devices. By utilising the Device Access Specification's matching algorithm with

specific codes describing the quality of a match, we can ensure that new drivers can be added to the system for similar devices without interference.

The current design provides for serial port drivers and a consuming driver for the Actigraph device. It allows expandability to support other classes of devices through the publishing of meta-data as OSGi service properties in the service registry. This means that new device classes (such as storage devices or sensor devices) can easily be integrated into the system as well.

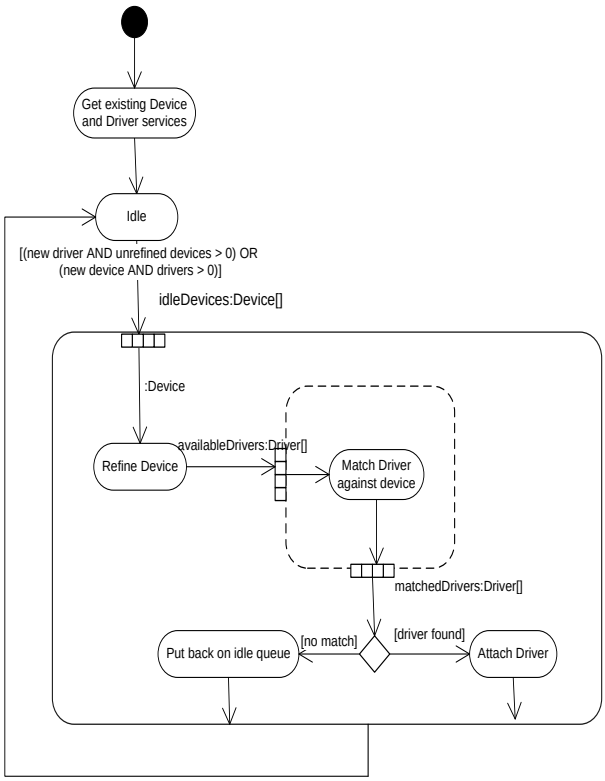

Fig. 4. Driver Detection

VI. Conclusion

We examined the requirements of a proposed system to obtain physiological and survey data from a real life environment in drivers for the purposes of understanding fatigue risk factors. Given the desire to conduct research with willing participants outside the laboratory, we investigated the need for a computer based device that could be placed in their homes to collect physiological and survey data and the requirements for such device to interface with their television, allow the connection of external medical monitoring devices and the secondary storage of acquired data sets. We also noted that this type of autonomous system for collection of such data in the field was a common theme in many areas of medical, biomedical research. We also established the need for middleware that could support remotely deployed gateways and manage attached devices and our choice of Java and OSGi to satisfy these needs.

We showed that our choice of a game console as a suitable hardware device over similar devices such as personal video recorders and set top boxes was due to their high availability and stable platform details.

The PlayStation-2 (PS2) was shown as a potential option because it could run Linux and allowed external hardware devices to be attached and was relatively inexpensive. However, the difficulty in attempting to compile software and run it on this platform, the difficulty of finding the needed second hand hardware, its poor performance, and lack of support, all demonstrated that the PlayStation-2 would be an inferior choice of platform. We chose the PlayStation-3 (PS3) as an alternative to PlayStation-2 because it could run Linux without having to modify the console or introduce an extra "Linux kit". It had greater hardware capabilities that increased the flexibility of the type of software that could be run and how this could be developed.

It was noted that whilst the PS3 was more expensive than the PS2 (including the extra hardware and Linux kit) and that PS3 was not expected to drop in price considerably soon, it had greater hardware availability, which may justify the extra price paid for its use in our research. At the time of writing this paper, the PS3 is our preferred computer system for the home for remote data recording and collection.

The process of elaboration and development of a flexible device management capability through a merger of OSGi and HAL software to enable the chosen platform PS3 for our gateway device to automatically manage peripheral devices plugged into the remote system (gateway) and connect the right applications to them was outlined.

A design was demonstrated in theory that utilises the HAL and OSGI Device Access Specification to provide a framework for device enumeration and events. This framework is extensible, and provides sufficient meta-data for distinguishing between devices. It also provides an automatic refining structure so that devices are automatically connected with drivers that can refine them and expose more specific device services. This work would benefit from future additions to HAL that detect new kinds of devices and provide better information about existing devices, provided the architecture of HAL doesn't significantly change.

The design was demonstrated through the specific development of a driver chain for the Actigraph that communicates over a virtual USB serial port using its own communications protocol that a third party sensor can be easily integrated into the gateway. By developing a generic serial port refining driver that refines serial port devices in the system and a specific Actigraph driver that layers on top of the serial port driver. The Actigraph driver uses the serial port device services to communicate with the Actigraph, read its status and download its current data log. This successfully demonstrated that the software and hardware combination that compose the gateway provide a platform flexible enough to support multi-level refining drivers for arbitrary devices that are automatically connected to the device when it is attached to the system, and conversely, disconnected from the device when it is detached from the system.

This choice of the Actigraph proved generic enough to provide a framework that would allow new devices to be managed by the system by merely adding the right drivers.

Lastly, a custom graphical application component was built

that uses the Actigraph device service to demonstrate that an application could be built atop this framework of devices services. The graphical application displays the status of the Actigraph and the status of any download tasks.

ACKNOWLEDGEMENT

The SmartData project was supported under an Australian Research Council grant (ARC Linkage grant No. LP0562407), Australia. We thank Venuganan Santhakumar for technical input.

REFERENCES

[1] Lal SKL & Craig A. 2005, 'Reproducibility of the spectral components of the electroencephalogram during driver fatigue', International Journal of Psychophysiology, vol 55(2), pp137-43.
[2] Ting P et al. 2008, 'Driver fatigue and highway driving: a simulator study', Physiology and behaviour, vol 94, pp 448-453.
[3] Li X. and Zhang W. 2004, 'The Design and Implementation of Home Network System Using OSGi Compliant Middleware', IEEE Transactions on Consumer Electronics, vol. 50, no. 2, May 2004.
[4] Zhang, H., Wang, F. and Yunfeng, A. 2005, 'An OSGi and agent based control system architecture for smart home', Proceedings of the 2005 IEEE Conference on Networking, Sensing and Control, pp13-18, Beijing, China, 2005.
[5] Kirchof, M. and Linz, S. 2005, 'Component-based development of Web-enabled eHome services', Personal and Ubiquitous Computing, vol. 9, issue 5, September 2005.
[6] Chan, M., et al. , 'Smart homes – Current features and future perspectives', Maturitas, 2009
[7] Chan, M.; Esteve, D.; Escriba, C. and Campo, E., 'A review of smarthomes – Present state and future challenges', Computer Methods and Programs in Biomedicine 91 (2008), pp 55-81, 2008
[8] Lin, C.; Young, S. and Kuo, T., 'A remote data access architecture for home-monitoring health-care applications', ScienceDirect Medical Engineering and Physics 29 (2007) pp 199-204, 2007
[9] Actigraph 2007, Actigraph GT1M Monitor / ActiTrainer and ActiLife Lifestyle Monitor Software User Manual, Actigraph LLC, March 2007, <http://www.theactigraph.com>.
[10] Van Hoff, A. 1997, 'The case for Java as a programming language', IEEE Internet Computing, vol. 1, issue 1, pp 51-56, Palo Alto, CA, USA.
[11] OSGi Alliance 2009, OSGi Alliance | About / The OSGi Architecture, viewed 16 March 2009, <http://www.osgi.org/About/WhatIsOSGi>.
[12] Apache 2009, Apache Felix, viewed 4 February 2009, <http://felix.apache.org/site/index.html>.
[13] Eclipse 2009, Equinox, viewed 4 February 2009, <http://www.eclipse.org/equinox/>.
[14] Knopflerfish 2008, Knopflerfish OSGi – open source OSGi service platform, view 4 February 2009, <http://www.knopflerfish.org/>.
[15] ProSyst 2009, OSGi Framework Implementations – open source Equinox and commercial – ProSyst, viewed 4 February 2009, <http://www.prosyst.com/products/osgi_framework.html>.
[16] Sentilla Corporation, Sentilla, 2009 <http://www.sentilla.com/>
[17] Sony Computer Entertainment 2009, Business Development/Japan | CORPORATE INFORMATION | Sony Computer Entertainment Inc., Sony Computer Entertainment, viewed 4 February 2009, <http://www.scei.co.jp/corporate/data/bizdatajpn_e.html>.
[18] Sinclair 2006, 'Sony stops making original PS', Gamespot, viewed 4 February 2009, <http://au.gamespot.com/pages/news/story.php?sid=6146549>.
[19] Microsoft 2000, Xbox Brings "Future-Generation" Games to Life, Microsoft Corporation, viewed 4 February 2009, <http://www.microsoft.com/presspass/features/2000/03-10xbox.mspx>.
[20] Sony 2001, EE Overview, Sony Computer Entertainment Inc., version 5.0, published October 2001, Tokyo, Japan.
[21] Playstation 2 Linux Community 2007, Linux for PlayStation 2 Community: Linux for Playstation 2 FAQs, viewed 4 February 2009, <http://playstation2-linux.com/faq.php>.
[22] Kaffe 2009, Kaffe.org, viewed 4 February 2009, <http://www.kaffe.org/>.
[23] Sony Computer Entertainment 2 2008, Linux Kernel Overview, viewed 19 March 2009, <http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-08.06.09/LinuxKernelOverview.html>
[24] IBM 2004, IBM developer kits for Java technology on Apple PowerPC hardware, viewed 19 March 2009, <http://www.ibm.com/developerworks/systems/library/es-apple.html>
[25] Freedesktop.org, freedesktop.org - Software/hal, 2009, <http://www.freedesktop.org/wiki/Software/hal>
[26] OSGi 2007a, OSGi Service Platform: Service Compendium, Release 4, Version 4.1, published April 2007, viewed 13 June 2009, <http://www.osgi.org>.
[27] OSGi 2009a, OSGi Service Platform Core Specification: Release 4 Version 4.2 (Public Draft 10 March 2009), The OSGi Alliance, viewed 07 June 2009, <http://www.osgi.org/Specifications/Drafts>.
[28] OSGi 2009b, OSGi Service Platform Release 4: Version 4.2 – Early Draft 3, published 7 March 2009, viewed 13 June 2009, <http://www.osgi.org/Specifications/Drafts>.
[29] Freedesktop.org, freedesktop.org - Software/dbus, 2009, <http://www.freedesktop.org/wiki/Software/dbus>
[30] Keane Jarvi, RXTX: The Prescription for Transmission, 2006, <http://users.frii.com/jarvi/rxtx/intro.html>