A General Framework For Motion Sensor Based Web Services

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Yiding Chai

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

With the development of motion sensing technology, motion sensor based services have been put into a wide range of applications in recent years. Demand of consuming such service on mobile devices has already emerged. However, as most motion sensors are specifically designed for some heavyweight clients such as PCs or game consoles, there are several technical challenges prohibiting motion sensor from being used by lightweight clients such as mobile devices, for example:

- There is no direct approach to connect the motion sensor with mobile devices.
- Most mobile devices don't have enough computational power to consume the motion sensor outputs.

To address these problems, I have designed and implemented a framework for publishing general motion sensor functionalities as a RESTful web service that is accessible to mobile devices via HTTP connections. In the framework, a pure HTML5 based interface is delivered to the clients to ensure good accessibility, a websocket based data transferring scheme is adopted to guarantee data transferring efficiency, a server side gesture pipeline is proposed to reduce the client side computational burden and a distributed architecture is designed to make the service scalable. Finally, I conducted three experiments to evaluate the framework's compatibility, scalability and data transferring performance.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Ralph Deters for hist continuous support and encouragement during my study. Dr. Deters provided me the inspiration of this research and give me numerous valuable advices during the progress.

I would like to thank my committee members: Dr. Julita Vassileva, Dr. Eric Neufeld and Dr. Daniel Teng for their valuable feedback and suggestions on my thesis.

I also would like to thank all MADMUC members for their interests and support about my work and experiments.

Finally, I would like to thank my parents and friends for their unconditional love and support.

Contents

List of Tables

List of Figures

List of Abbreviations

CCD	Charge Coupled Device
CORBA	Common Object Request Broker Architecture
FIFS	First In First Serve
FPS	First Person Shooter
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
IDL	Interface Description Language
MCMC	Markov chain Monte Carlo
REST	Representational State Transfer
RMI	Remote Method Invocation
ROA	Resource Oriented Architecture
RPC	Remote Procedure Call
RTT	Round Trip Time
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Accessing Protocol
UDDI	Universal Description Discovery and Integration
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WS	Web Service
WSDL	Web Service Definition Language
XML	Extensible Markup Language

Chapter 1

Introduction

Along with the rapid development of motion sensing technology, various new applications and market opportunities for motion sensors have emerged in recent years: From the accelerometer and optical sensing technology used in the controller for Nintendo's Wii console, to the multi-touch screen and gyroscopic sensor embedded in the Apple iPhone, motion sensors are not only used in specialized industrial applications but have also grown to embrace a much broader market including personal mobile devices, sporting equipments, medical rehabilitation and security monitoring systems. On November 4, 2010, Microsoft launched a new peripheral for its Xbox 360 console named Kinect, which indicates another remarkable development in motion sensor technology: It's the first depth camera based motion sensor for the consumer market that is able to track player's full body motion in three dimensions without any accessories attached(See Figure 1.1).

Figure 1.1: Kinect Motion Sensor [1]

Designed as a game controller, Kinect frees players from traditional game input devices by reflecting their movements into the 3D virtual environment in real time. This device brings a new gaming experience and makes numerous innovative game designs possible. Besides that, as a good range camera at an affordable price, Kinect has also inspired lots of innovative applications outside the game world: Numerous third party developers and organizations have explored many potential applications of Kinect beyond the device's intended purpose of playing games. Here are some representative examples:

- By conducting image processing on Kinect depth data [2], a group of students from MIT CSAIL built

a motion-controller user interface that is very similar to the one envisioned in the science fiction movie Minority Report.

- Alexandre Alahi, a PHD student from EPFL invented a video surveillance system using multiple Kinect devices to keep track of a group of people in complete darkness [3].
- Researchers from the University of Minnesota adopted Kinect to measure a range of disorder symptoms in children such as autism, attention deficit disorder and obsessive compulsive disorder [4].

Inspired by so many creative applications of the motion sensor and the rapid development of the smart mobile device market, it is interesting to explore if it is possible to use Kinect-like motion sensor functionalities on mobile devices. Once it became possible, motion sensors could be applied into a much wider range of areas and more potential applications could be generated:

- Remote Home Surveillance. The automatic human movement detection feature makes the motion sensor a good candidate for home security. Once the sensor is able to communicate with mobile devices remotely, it could be configured to send alert notification to subscribed mobile devices when unauthorized entry is detected.
- Remote Medical Service. Once the motion sensor can be accessed as a web service, distance medical exam & diagnose will become possible. To examine patients with movement disorder, doctors can remotely define a set of movement sequences and ask patients to perform these movements in a motion sensor monitored room. Then the patients' movement data can be extracted by those sensors and sent back to the doctor location for further analysis.
- Enhanced Mobile Game Control. When a motion sensor is able to communicate with mobile devices, it can serve as a game controller for mobile devices: Information such as player's posture and location could be instantaneously reflected in mobile games. For example in an First Person Shooter(FPS) game on mobile device, the game's first-person view could be completely determined by the position and orientation of real player's head, which will make players feel they are exploring the virtual world by themselves.

A straightforward way to make motion sensors usable on mobile devices is to adopt a traditional host-peripheral approach, e.g. plugging a motion sensor (as the peripheral) into a mobile device (as the host) via a physical connection (USB cable) or a short range wireless connection (RF or Bluetooth). After that the host can interact with the peripheral device through a pre-installed device driver for the motion sensor. However, this approach has several limitations:

- First of all, given the diversity of the hardware/software of mobile devices, designing and implementing a local driver for each kind of host devices is impractical. Taking Kinect for example, although there are already many open source drivers for Kinect like OpenNI [5] and OpenKinect [6], building those drivers from source code on various mobile platforms is challenging.

- Secondly, most motion sensors are high level sensors that consume considerable computational power: Taking Kinect for example, to obtain the skeleton data, complex human body detection algorithm need to be performed on each depth frame at 30 times per second, which is a very computational expensive task that is beyond the average computational capacity of current mobile devices.

For the above reasons, using a motion sensor as a local input device is neither elegant nor practical. So instead of trying to connect the motion sensor with individual mobile device directly, an alternative approach is to host a motion sensor on a web server and expose its functionalities as a web service. (See Figure 1.2).

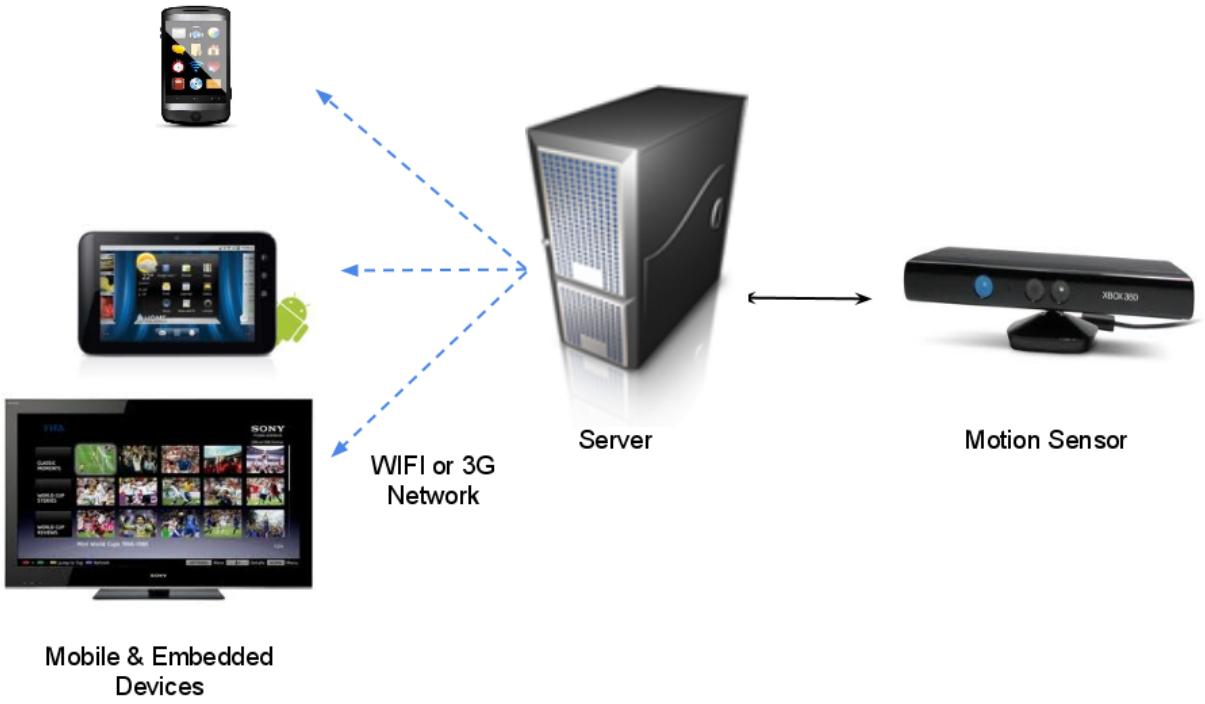


Figure 1.2: Host motion sensor as web service [7] [8] [9]

In such a web service approach, several previous problems can be solved:

- First, the client devices are freed from physical connections or distance constrains: They could access the service from anywhere as long as they have connection to the Internet.
- Secondly, the server layer eliminates the need of implementing a motion sensor driver for each type of mobile device. By defining a unified communication protocol that is supported by most of the mobile devices, the server itself could serve as an interface between the motion sensor and its clients, through which various client devices can access the service (sending requests and retrieving results) in a unified way.
- Finally, the client-server architecture could help to migrate a large portion of computational work from the client side to the server side, which might significantly reduce the requirement of computational capacity for the clients therefore making the motion sensor usable by more devices.

Chapter 2

Problem Definition

The main goal of this research is to design a framework to publish the motion sensor functionalities as a web service that is accessible by hybrid apps on mobile devices via HTTP connections (See Figure 2.1).

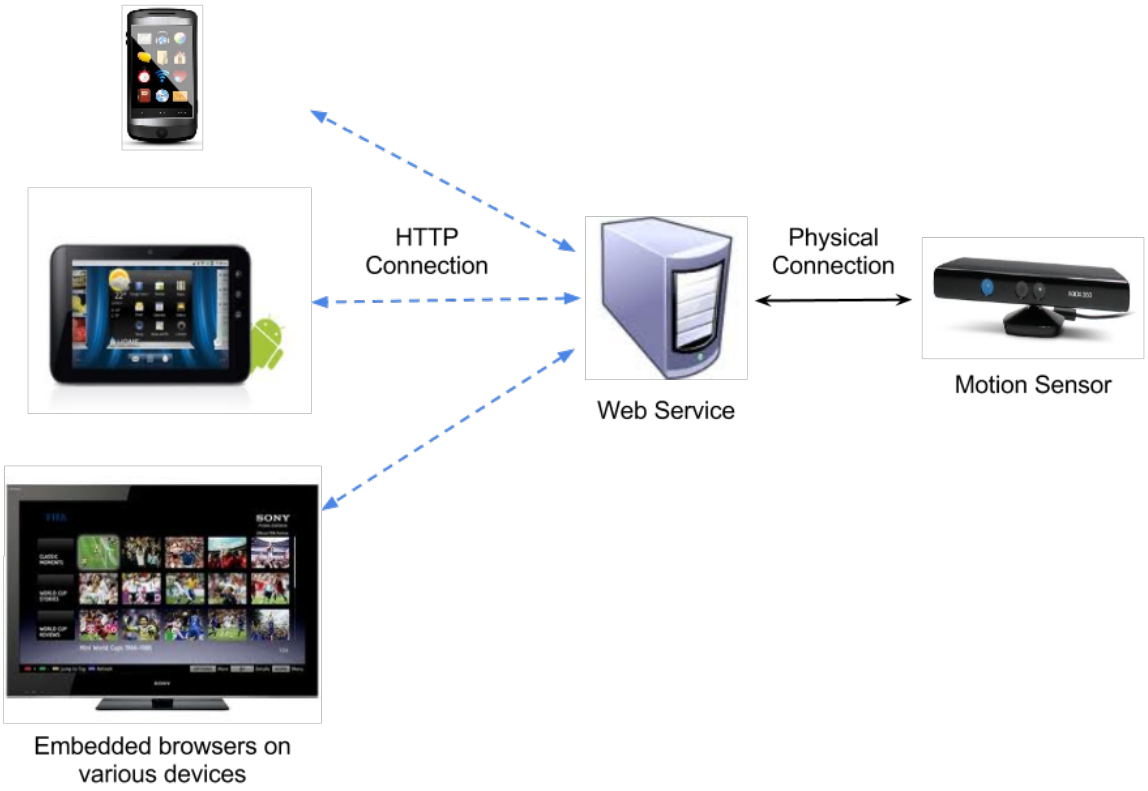


Figure 2.1: Overview of Motion Sensor Web Service [7] [8] [9]

To clarify the problem definition, some terms need to be specified: First, the kind of the motion sensors used in my research is a human posture sensor that is based on range imaging, which is essentially a composite of multiple sensors including:

- A video camera that can capture photographic data (in form of RGB image) of the scene;
- A range camera that can capture depth data (in form of depth image) of the scene;

- A human body detection pipeline (either software or hardware) that is able to extract human body information from the data generated by the above two sensors and describe its posture in an abstract form (skeleton).

All data output by the above components are in form of streams refreshing at high rate. And the functionalities that the motion sensor web service is supposed to provide include:

- Enabling clients to access all the three data streams output by the motion sensor (RGB, depth, skeleton).
- An additional gesture recognition pipeline accompanied by a gesture event subscription & notification module, which enables clients to create and subscribe to gesture events and get instant notifications when they are triggered.

In the rest part of this chapter, I will analyze some key issues that need to be solved in designing such a web service framework from the perspectives of both service consumer and service provider.

2.1 Critical Issues from the Perspective of Service Consumer

- **Client Side Computational Burden.** A motion sensor is usually a high level sensor, which needs a series of computational intensive post processing steps (for example skeleton extraction and gesture recognition) to make its raw outputs usable. However our targeted clients – embedded web browsers on mobile devices – usually have very limited computational capacity. Therefore, the first challenge is: **How to design an architecture that migrates most of the computational intensive tasks from the clients side to the server side?**
- **Bandwidth Consumption.** Most motion sensors are high-speed data generators refreshing data frames at high rates (normally 30 frames per second for each stream). While my targeted clients – mobile devices are usually on data plans with limited bandwidth and relatively expensive data rates. With regard to this contradiction, the second challenge I need to deal with is: **How to design a data transfer scheme that is able to transfer streams of data to clients through the HTTP protocol with good bandwidth efficiency?**

2.2 Critical Issues from the Perspective of Service Provider

- **Scalability.** How well a web service scales as the number of clients increases always concerns the service provider, especially when the service provides data in form of streams rather than in traditional HTTP messages. In addition to that, deciding a web service's serving capability is usually a hard problem for the service provider: A web service system which is able to serve a large number of clients usually wastes lots of energy when the actual number of clients is low, while a more energy-economical system may not be capable enough to meet all the demands when clients number peaks. So the third

challenge I face is: **How to design an architecture that is potentially capable of serving large numbers of clients that can be dynamically adjusted according to the actual workload?**

- **Extensibility.** After a web service is launched online, it is common that the service provider wants to put additional functions onto the existing ones to improve the service. To facilitate such a process is my last challenge i.e. **How to design and implement a core data processing module in my service to make it easy to extend?**

Chapter 3

Literature Review

This chapter reviews works on the related issues and technologies. First I will look into some previously conducted research projects on using the sensor-based web service in section 3.1. Then I will give a brief introduction into the technologies behind human motion sensors in section 3.2. After that I will compare and discuss several types of web services that might be used to build the motion sensor web service in my research in section 3.3. In the end a summary will be presented in the section 3.4 to discuss some design decisions and possible solutions to questions proposed in problem definition.

3.1 Related Works on Sensor based Web Services

Along with the rapid development of sensor technology, sensor-based web services have been drawing attention from both industry and academy in recent years. Their ability to collect accurate and reliable information makes them perfect tools to conduct long term scientific researches and observations in order to make early warnings and quick responses to potential disasters or threads. Representative research projects in this field include:

- The habitat monitoring project [10] was conducted by researchers from Intel Research and UC Berkley, in which they proposed an framework of utilizing sensor network to monitor a habitat. To demonstrate their idea, they deployed 30 sensor nodes a on a small island in Maine. Each of these sensors is an composite of multiple sensors to measure temperature, air pressure, humidity and so on. All the sensor nodes are connected to a base station which allows its users to obtain live data from a website at http://www.greatduckisland.net.

- The air pollution monitoring project [11] was conducted by researchers from the Imperial College London, which proposed a GRID Infrastructure that uses distributed sensor grids to collect airborne pollutants data. Each sensor samples every two seconds and stores the data in a data warehouse accessible by SQL queries for public search.

- The soil moisture monitoring project [12] was conducted by researchers from the University of Western Australia. The purpose of the project is to provide useful soil moisture data that enables people to make dynamic responses to rainfalls. Several sensors to measure soil moisture and rainfall level are

deployed, which are sampled every twelve seconds. Whenever any of their measured values exceeds a predefined threshold, related event will be generated and sent out through the network.

- The wireless sensor networks for home health care project [13] was conducted by researchers from the UC Berkeley and Intel Corporation. In this project, they have developed several prototypes to demonstrate some applications of sensor web services in home health care, which include: Using a 3-axis accelerometer sensor attached to infants' pajamas to detect their sleeping position; Using an acoustical sensor that communicates with the attached vibrators or LEDs to warn hearing impaired users of some critical audible situations; Using wearable blood pressure sensor to send users' blood pressure and pulse data to the their personal computers, where those information will be stored for further analysis.

However none of the above sensor based web services can be directly applied to my case:

- Firstly the sensors used in the above researches are mostly low-level ones such as temperature, humidity or pressure sensors. They can generate outputs without any significant post processing. Therefore in most of those projects, the sensors are deployed standalone without being accompanied by any processing unit.
- Secondly in the above works, the data gathered by the sensors is usually used for long term analysis. So the sampling rate is usually low (at most one time per two seconds) that a normal poll sampling method is fast enough to meet the needs. However in my case, some time-critical client applications need to retrieve data at a high rate, which makes the sampling models used by above researches unsuitable.
- And thirdly the targeted service consumers of the above works are general computers for research purposes, which are not resource constrained like my targeted consumers – mobile or embedded devices.

3.2 Human Body Motion Tracking System

Broadly speaking, a motion sensor is a device that contains a physical mechanism or electronic sensor that quantifies motion in its field of view. According to the detection method used, motion sensors can be classified as sound based, magnet based or reflection of transmitted energy based. In this research, I focus on a much narrower range of motion sensors, namely human posture tracking system, which is capable of recognizing and tracking human body movement in three dimensional space. Currently these kind of sensors are often used as peripheral input devices for computers, which enable the user to interact with the computer solely through their natural body movements, without the intervention of any other traditional input devices such as keyboard, mouse...etc. Without loss of generality, in this section, I will use the Kinect as an example to illustrate the technologies behind contemporary motion sensors. As mentioned in the previous chapter, a motion sensor is usually made up of three key components: a RGB camera, a range camera and a posture extraction pipeline.

3.2.1 Structured Light Based Range Camera

Range scanning technology has been developing very fast in recent years. A complete overview is available in Bernardini et al.[14] and Curless et al. [15]. According to Curless et al. [15], a common characterization subdivides those devices into two categories: contact and non-contact (see Figure 3.1). For real time human body motion tracking, there is a very important subclass of the non-contact methods, which is based on optical technology. Unlike the ones in the contact category, the optical based method can be completed in real time. In addition to that, by using a depth sensing method similar to the human vision system, the optical based method neither needs any expensive special devices nor does it any harm to the human body as industrial CT, and yet can provide more accurate result than microwave radar or sonar based approach. These features make it very suitable for home entertainment use.

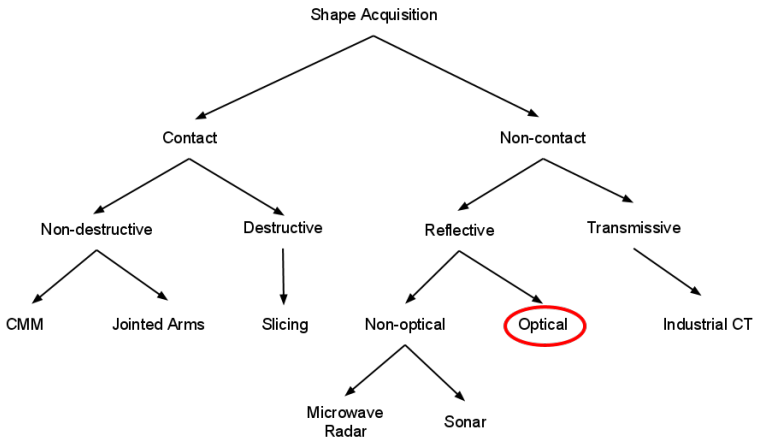


Figure 3.1: Taxonomy of shape acquisition[15]

The optical based method can be further divided into two approaches: active and passive, as shown in figure 3.2. Although these passive methods such as stereo and shape from shading is undergoing rapid development in recent years, as of today they are still very sensitive to noise, color and light variation, which makes them not robust enough to produce accurate depth data for motion analysis in a general situation. For the above reasons, I will focus on the structured light based active approach.

A typical structured light based active optical device consists of an emitter, which projects a predefined type of structured light pattern onto the objects to be scanned, and a sensor, which is typically a Charge-Coupled Device (CCD) that captures the distorted patterns reflected by the surfaces of the objects. When a one-to-one correspondence between the original pattern and the reflected pattern is established, the sample point's 3D coordinate can be reconstructed by a method named triangulation (see Figure 3.3), given the sample point's coordinate on the emitter plane, the position of sensor plane and the distance between the center points of emitter and sensor.

Traditionally, the projected patterns are rectangle grids or line arrays in visible light, but when it comes to the case of Kinect, the above solution is not feasible: To make the projected pattern recognizable for

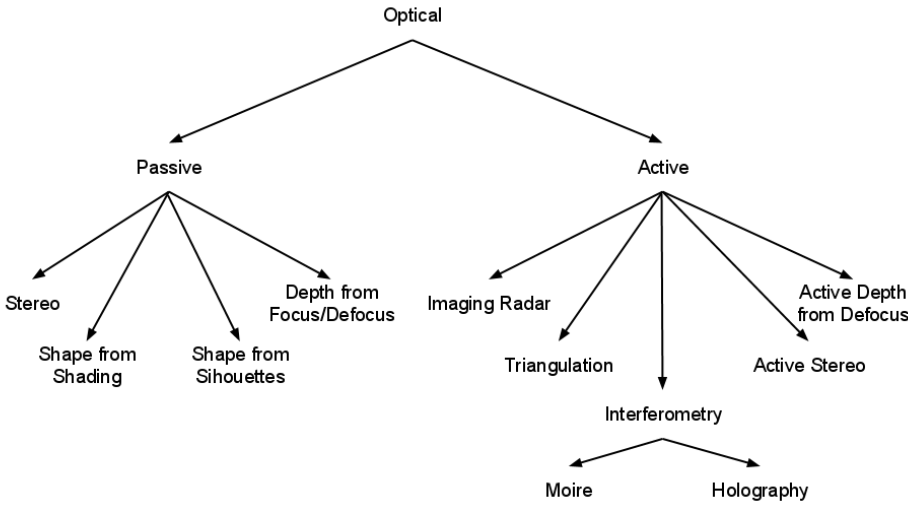
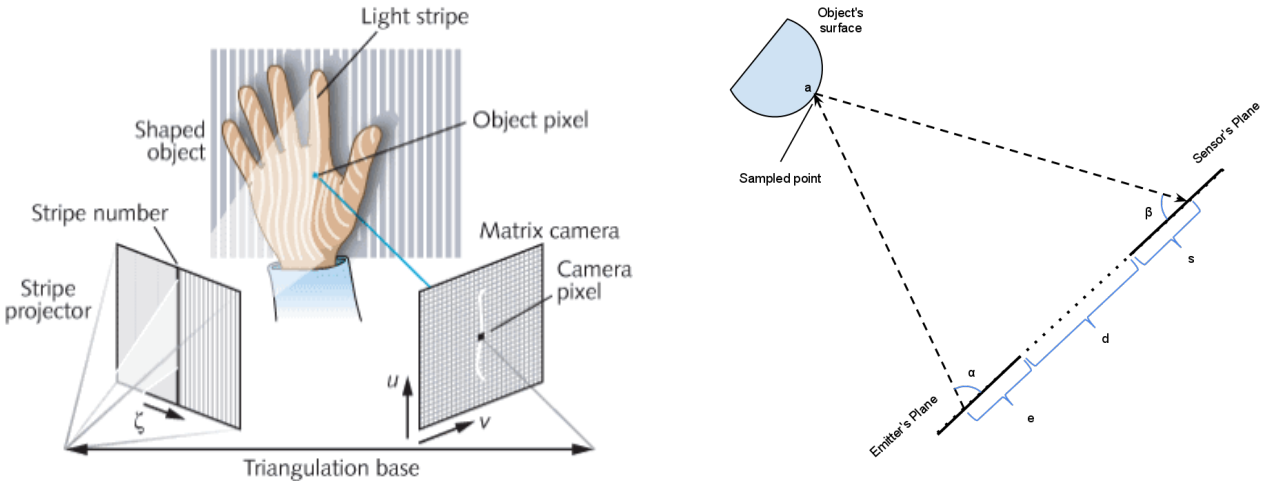


Figure 3.2: Taxonomy of optical approach [15]



(a) [16] **(b)**

Figure 3.3: Structured Light Projection and Triangulation

the sensor, a very powerful laser projector is needed to maintain high signal-to-noise ratio, which has the potential to damage the user's eyes. In order to avoid that, PrimeSense (Tel Aviv, Israel, the Kinect range camera provider) adopts two techniques:

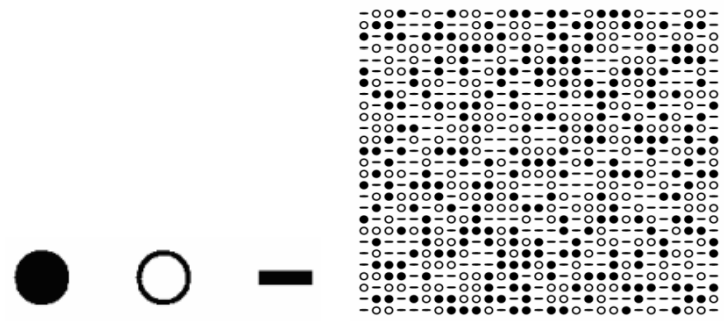
1. First, instead of using visible light, the emitter in the Kinect projects infrared structured light in the 800 to 900 nm range (see Figure 3.4), which is invisible to human's eyes but in a range where the CCD sensor has high sensitivity. As a result, the users will not be disturbed by the light patterns it projects. Moreover, as the CCD sensor only measures the infrared light in such a range, Kinect becomes insensitive to visible light: Normal environmental light changes will not significantly affect the quality of the depth image captured by the range sensor, which is a desirable feature for the following human

pose recognition procedure.

Figure 3.4: Kinect's infrared structured light captured by its infrared light sensor

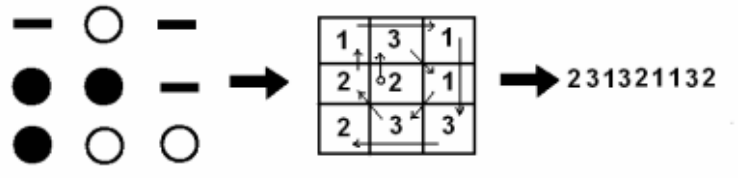
2. Secondly, in order to use triangulation to determine a point's depth, a one-to-one correspondence between the original pattern (undistorted) and the reflected pattern (distorted) must be established (see figure 3.3). To make the process easier and more accurate, PrimeSense uses a proprietary light projection pattern based on light coding, which is invented by Albitar et al. [17] and Young et al. [18]. While the specific pattern is beyond our scope, the general idea of light coding can be briefly described as following: By defining a set of primitive shapes (see figure 3.5a) and combining them in a special way, we can get a matrix like pattern (see figure 3.5b), where every element (in this case, a window formed by 3×3 primitives) has a unique combination, which can be decoded to a unique sequence of numbers (see Figure 3.5c) that will be used as the element's ID number. Then, for each element in the reflected pattern received by the sensor, it will be matched to the element in the original pattern that has the closest ID number to its own ID (measured in hamming distance) (see Figure 3.5c).

As shown in Figure 3.6, when the reflected pattern is received by the CMOS sensor, the generated lighting code image will be sent as input into a processing chip (PS1080 SoC), where the original and reflected pattern will be corresponded, and the triangulation process will be conducted to compute the depth value for each pixel. Finally, the depth value will replace the light intensity value stored in each pixel of the lighting code image, which will be output as a depth frame.



(a) The considered primitives: disc, circle and stripe

(b) The structured light pattern



(c) The decoding process

Figure 3.5: Light Coding Technology [17]

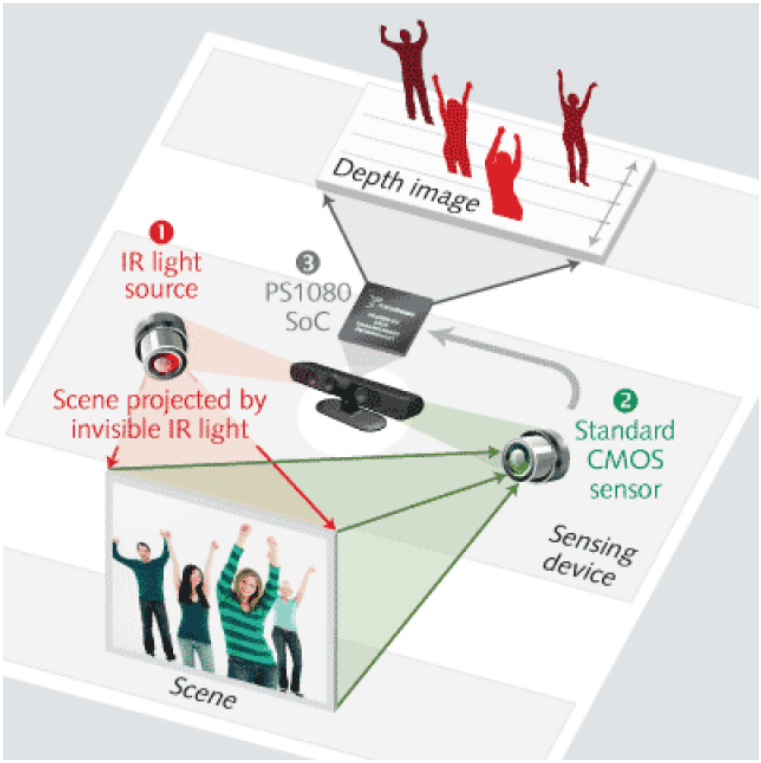
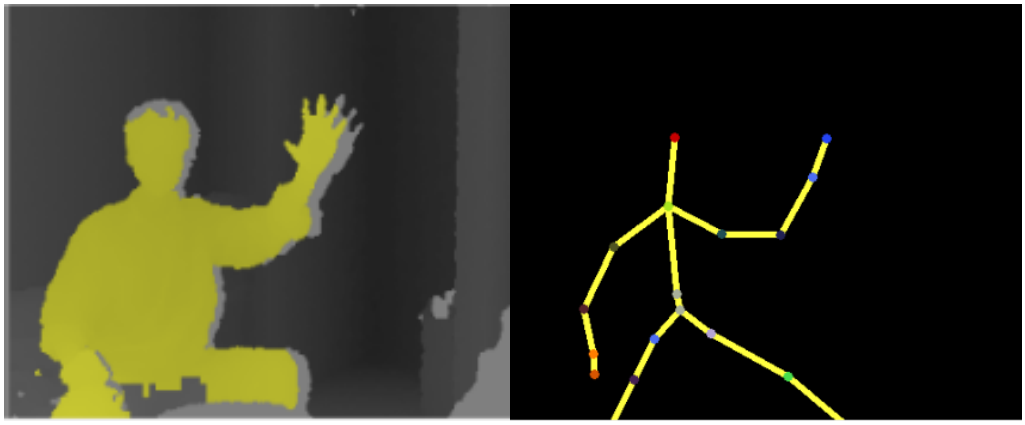


Figure 3.6: PrimeSense Range Camera Working Scheme [19]

3.2.2 Real Time Human Pose Recognition From Depth Image

After the depth frame of the scene is generated, it can't be directly used without extracting useful human body information. Human pose estimation is a process to detect human body from an image and further derive an abstract representation of the body's posture(See Figure 3.7).



(a) Depth Frame **(b)** Human Posture Abstraction (Skeleton)

Figure 3.7: Human Posture Estimation

In the computer vision community, human pose estimation is a research topic generating numerous works, Moeslund et al. [20] and Poppe et al. [21] give good reviews about them. Compared with traditional image frame generated by RGB camera, depth frame generated by range camera can be naturally transferred into 3D space. Moreover it is more independent of the variations in light and perspective. So it is considered as a more suitable basis for human pose estimation.

For motion sensors used for interactive purposes, human pose estimation needs to be performed in real time on each depth frame generated by the range camera, which requires the algorithm fast enough to be able to keep up with the camera refreshing rate. In this chapter, I will focus my review on a paper published by Microsoft Research team in 2011 [27] and give a brief introduction about the approach used by the Kinect to extract the posture information out of depth image in real time.

Their method contains three steps: First, on a human body region in depth frame, the algorithm performs a dense per-pixel classification task to segment the whole body into several parts, which is color coded in figure 3.8. Then, for each body part, the algorithm proposes a skeleton joint position based on positions of all pixels belonging to the part. And after that these joints are combined with the predefined connectivity relationships to form the structure of the posture abstraction (skeleton).

Decision Tree Based Body Parts Labeling

Body parts labeling is a process to classify each pixel in depth frame into a specific body part. To complete this task, some features that can distinguish pixels of one part from pixels of other parts need to be found.

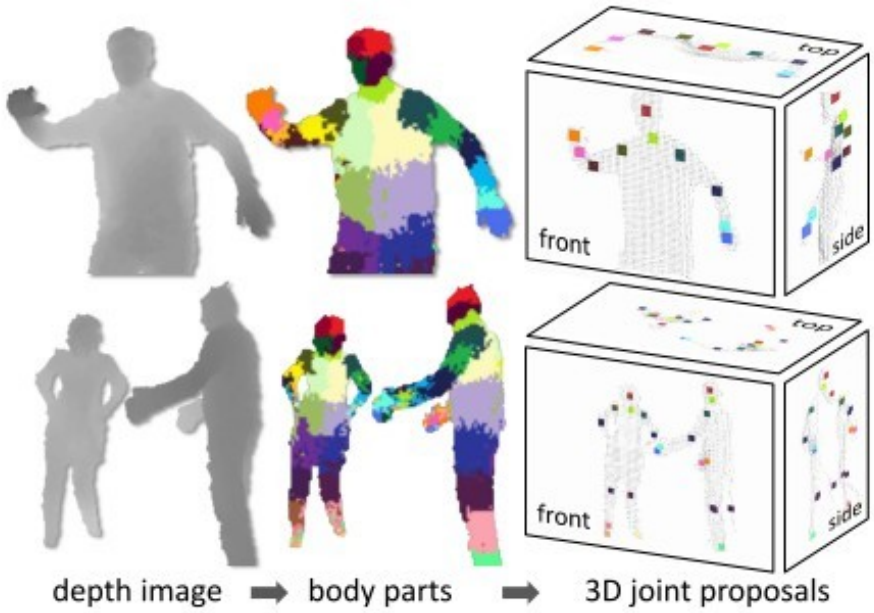


Figure 3.8: Overview of Skeleton Generation. From a single input depth image, a per-pixel body part distribution is inferred. (Colors indicate the most likely part labels at each pixel, and correspondent with the joint proposals). Local modes of this signal are estimated to give high-quality proposals for the 3D locations of body joints, even for multiple users [27].

For this purpose, in Shotton et al. [27], they use a series of depth comparison features computed by:

$$f_\theta = d_I(x + \frac{u}{d_I(x)}) - d_I(x + \frac{v}{d_I(x)}), \tag{3.1}$$

where $d_I(x)$ is the depth at pixel x in frame I, and $\theta = (u, v)$ describes two offset vectors u and v, which is normalized by $\frac{1}{d_I(x)}$ to ensure depth invariance. This feature computes the depth difference between two pixels with offsets $\frac{u}{d_I(x)}$, $\frac{v}{d_I(x)}$ to pixel x respectively. As illustrated in figure 3.9, a feature as f_{θ_1} will give a great response to pixels near horizontal edges such as the top of the body where depth value changes dramatically along vertical direction, while feature f_{θ_2} looking for long thin structures aligned in a certain degree such as arms and legs.

Individually these features only serve as very weak clues about which part the pixel belongs to, but when a series of such features with different θ is combined in a randomized decision forest, it will become a powerful classifier to disambiguate all predefined parts of a body [27]. As shown in figure 3.10, a decision forest is made up of T decision trees and each of them are trained on a large number of labeled synthetic body depth images (detailed synthetic image generation and decision tree training method is described in section 3.3 of Shotton et al. [27]). Each decision tree consists of several split nodes and leaf nodes. In each split node, there is a feature specification θ and a threshold τ. To classify a pixel x, for each tree, one starts from the root and repetitively evaluates formula 3.1 on split node, then branching either left or right according to the result of comparison to the threshold τ until reaching a leaf node, where a pre-learned distribution $P_t(c|I, x)$ over the body part label c is stored. After the above processes for all T trees in the forest are completed, T

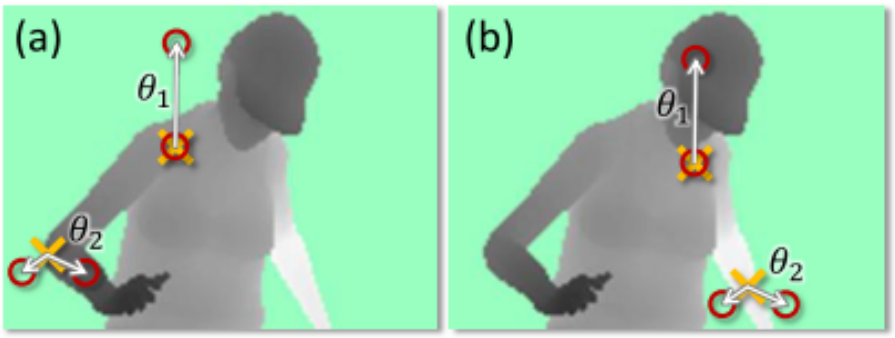


Figure 3.9: Depth image features. The yellow crosses indicate the pixel x being classified. The red circles indicate the offset pixels as defined in Eq. 1. In (a), the two example features give a large depth difference response. In (b), the same two features at the new image locations give a much smaller response [27].

conditional possibility distributions $P_t(c|I,x)$ will be derived and combined together as one final distribution:

$$P(c|I,x) = \frac{1}{T}\sum_{t=1}^{T} P_t(c|I,X). \tag{3.2}$$

The class c_n with the greatest possibility in $P(c|I,x)$ will be selected to label x.

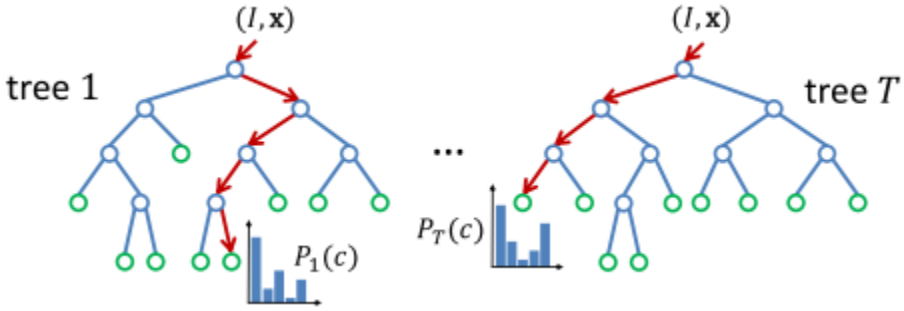


Figure 3.10: Randomized Decision Forest. A forest is an ensemble of trees. Each tree consists of split nodes (blue) and leaf nodes (green). The red arrows indicate the different paths that might be taken by different trees for a particular input. [27]

Mean Shift Based Joint Position Proposal

The output of the previous step is a segmented human body depth image containing per-pixel information about body parts (See the middle picture of Figure 3.8). To generate a more concise description of the posture as a skeleton, the position of every segment needs to be proposed as a single joint point by pooling across all the pixels of that segment. To complete this task, the pooling method they [27] adopt is a non-parametric analysis method called mean shift. Briefly speaking, the mean shift method treats all discrete data as samples from an function of empirical probability density, and uses an iterative procedure to locate the local maximal points of the function:

Given n d dimensional sample points $x_i \in R^d$ and a kernel function $K(x)$ which satisfies:

$$\int_{R^d} K(x)\mathrm{d}x = 1, \tag{3.3}$$

where

$$K(x) \geq 0(\forall x \in R^d). \tag{3.4}$$

For each sample point x_i, a kernel center x is initially placed on it and iteratively updated by the following process:

1. The weighted mean of the density over kernel centered at x is computed as:

$$m(x) = \frac{\sum_{i=1}^{n} K(x_i - x)x_i}{\sum_{i=1}^{n} K(x_i - x)}. \tag{3.5}$$

2. $m(x)$ is updated to x: $x \leftarrow m(x)$.

3. Iterate step 1 and 2 until convergence, i.e., $x - m(x) = 0$.

The above procedure is proven to converge by Comaniciu et al. [28], and if the kernel function is reasonably selected, the path defined by successive mean shift iterations will finally culminate to the nearest local density maxima, regardless of the starting point (See Figure 3.11). In this way, the mean shift can be used as a clustering algorithm, which groups all points whose paths ending up on the same local maxima into one cluster (See Figure 3.12).

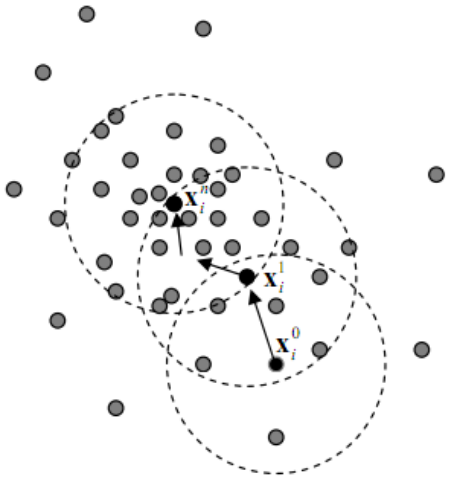


Figure 3.11: Mean shift procedure. Start at data point x_i and run the mean shift procedure to find the stationary points of the density function. Superscripts denote the mean shift iteration, the shaded and black dots denote the input data points and successive window centers respectively, and the dotted circles denote the density estimation windows [29]

Figure 3.12: Mean Shift Clustering. (a) Two-dimentional source data set of 110400 points (b) Clusters (rendered in different colors) obtained by mean shift [28]

When it comes to determining the joint position in depth image, the kernel function used by Shotton et al. [27] is defined as:

$$K_c(\hat{x}) \propto \sum_{i=1}^{N} w_{ic} exp\left(-\left\|\frac{\hat{x}-\hat{x}_i}{b_c}\right\|\right) \tag{3.6}$$

Where c denotes the classification of the part, $\hat{x}$ is a 3-dimensional world space coordinate, N is the number of image pixels in depth frame, b_c is a learned per-part bandwidth, $\hat{x}_i$ is the 3-dimensional world projection of pixel x_i in the depth frame, w_{ic} is a pixel weighting parameter defined as:

$$w_{ic} = P(c|I, x_i) \cdot d_I(x_i)^2. \tag{3.7}$$

This parameter takes into consideration of both the inferred body part probability $P(c|I, x_i)$ and the world surface area covered by the projection of pixel x_i (which is in proportion with $d_I(x_i)^2$) to ensure these density estimation values are depth invariant.

To propose a joint position for part c, every pixel whose pre-learned probability $P(c|I, x_i)$ is above certain threshold λ_c will be used as a starting point of the mean shift process. When the processes for all pixels are finished, they will converge on a few local maximal points. For each local maximal point, a confidence estimate will be computed as a sum of all pixel weights reaching it. Finally the local maximal point with the greatest confidence estimate will be proposed as joint position for part c.

3.3 Web Service

According to W3C definition [30], **Web Service (WS)** is "a software system designed to support interoperable machine-to-machine interaction over a network" [30], which can be implemented in many different styles and approaches. According to the communication method and main architecture used, web services can be broadly divided into three categories: **Remote Procedure Call (RPC) Based WS**, **Service Oriented Architecture (SOA) Based WS** and **Representational State Transfer (REST) Based WS**. As RPC usually maps its services directly to language or platform specific methods, which makes it unsuitable for web services targeted at a wide range of mobile platforms, this review will be focused on SOA and REST.

3.3.1 SOAP Based SOA WS

To overcome the problems of RPC approaches such as CORBA and DCOM, Gartner, an information technology research and advisory company, introduced a new design principle named Service Oriented Architecture (SOA) in 1996, which is now considered as the most promising approach to build loosely coupled systems [31]. The OASIS SOA Reference Model group [32] defined it as "a paradigm for organizing and utilizing

distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations" [33]. By emphasizing on standardization, interoperability and reusability, SOA enables WSs to be used as building blocks regardless of their computing platforms and organizations.

In a typical SOA scenario, there are three types of participants involved:

- **Service Provider**: Servers that actually host the services
- **Service Requester**: Client machines looking for some services
- **Service Registry**: A software agent lists all services provided by service provider to make them easy to be discovered by service requesters.

Between those three types of participants, a unified web service protocol stack is agreed to enable communication, which contains four protocols:

- **Transport Protocol**: responsible for transporting messages between network applications, which can be any one of HTTP, SMTP, FTP, as well as the more recent Blocks Extensible Exchange Protocol (BEEP).
- **Messaging Protocol**: responsible for encoding messages in a common XML format so that they can be understood on either side of a network connection. Currently, the messaging protocol can be any one in XML-RPC, WS-Addressing or SOAP.
- **Description Protocol**: used for describing the public interface to a specific Web service. The WSDL interface format is typically used for this purpose.
- **Discovery Protocol**: used for centralizing services into a common registry so that network Web services can publish their location and description together, and the clients can easily discover what services are available on the network. Universal Description Discovery and Integration (UDDI) is invented for this purpose, but it has not been widely adopted.

In SOA, the service provider describes the functionalities provided in WSDL, and publish the WSDL document to a public service registry. And then the service consumers can use SOAP message to interrogate the registry for the WSDL document. After that, according to the method signature information in the WSDL document, service consumers are able to communicate with the service provider and call any of the operations listed in the WSDL file by sending SOAP messages in an agreed format (See Figure 3.13). In the following of this section, we will give detailed reviews on WSDL and SOAP technologies respectively.

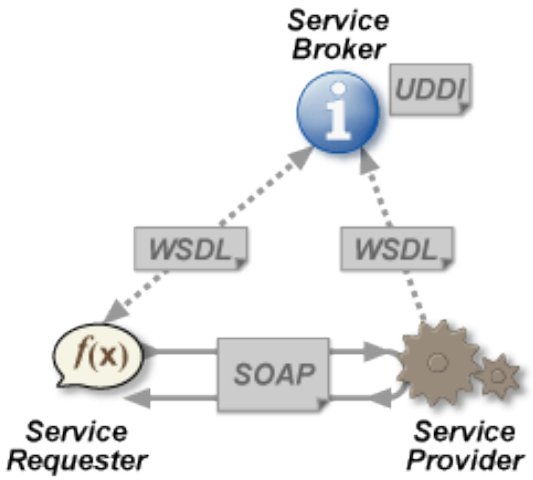


Figure 3.13: A Overview of SOA Web Service [34]

WSDL

WSDL is an **Extensible Markup Language (XML)** based language providing machine-readable description of the functionalities provided by a web service, which includes a list of functions and their input/output parameters. Its initial version – WSDL 1.0 was published by IBM, Microsoft and Ariba for their SOAP toolkit in 2000 [35]. One year later, WSDL 1.1 came out as a formalization of WSDL 1.0 [36], which is still widely in use. To accommodate the rapidly evolving WS Architecture style, a new version – WSDL 2.0 started to be drafted by W3C in 2003, and became a W3C recommendation on 2007 [37]. As WSDL 2.0 is still in a draft version as of July 2012, this review is based on WSDL 1.1. A WSDL 1.1 document usually has five major elements, as listed in table 3.1:

Element	**Description**
< *types* >	A container for data type definitions used by the web service
< *message* >	A typed definition of the data being communicated
< *portType* >	A set of operations supported by one or more endpoints
< *binding* >	A protocol and data format specification for a particular port type
< *service* >	A set of addresses specify where a bound operation may be found

Table 3.1: WSDL elements

All elements except < *types* > can be defined multiple times in a document. And they are arranged in a fixed order as shown in the following sample code:

```
<definitions>
     <types>?
        <!-- Defines the XML types used in the WSDL -->
     </types>
     <message>*
        <part element="..." or type="..."/>*
```

```
        </message>
        <portType>*
            <!-- Defines the web service "methods" -->
            <operation>*
                <input message="..."/>?
                <output message="..."/>?
                <fault message="..."/>*
            </operation>
        </portType>
        <binding>*
            <operation>
                <!-- Binding of the operation to a protocol, e.g. SOAP -->
            </operation>
        </binding>
        <service>*
            <port name="..." binding="...">
                <!-- Specifies the address of a service,
                e.g., with soap:address -->
            </port>
        </service>
    </definitions>
```

WSDL 1.1 is designed to be highly reusable. One can define a message that can be used by multiple operations (porttypes), and each operation can be bound to different URLs. But such flexibility also makes WSDL 1.1 document kind of verbose and not reading-friendly to human.

SOAP

SOAP, originally defined as **Simple Object Access Protocol**, is a protocol designed for exchanging structured information between different platforms. Its first edition was designed by Dave Winer, Don Box, Bob Atkinson and Mohsen Al-Ghosein for a Microsoft project in 1998. It first appeared in public as an IETF draft in November 1999 [38]. Shortly after that, in December of 1999, SOAP 1.0 was released. In May of 2000 the 1.1 version was formally submitted to the W3C [39], and very soon it became the core of the emerging Web Services technologies. The current version is 1.2, which became a W3C recommendation on May, 2003 [40] (The examples given in this section are all in SOAP 1.2).

Like WSDL, SOAP messages also rely on XML, and can be transported through a wide range of application layer protocols including **Hypertext Transfer Protocol (HTTP)** and **Simple Mail Transfer Protocol (SMTP)**, which means SOAP messages can be easily tunneled through firewalls and proxies. Besides that, SOAP is designed to be extensible from the very beginning. By leveraging the power of XML, various third-party standards can be integrated into it, which are often referred as WS-*, for instance WS-Security, WS-routing and so on. Despite its complicated extensions, the basic structure of SOAP is very light-weighted. As shown in Figure 3.14, a basic SOAP message only contains four elements:

- An application layer protocol header that wraps the whole message (can be a header of any supported protocol, depending on which one is used, in figure 3.14 it's a HTTP header).
- A SOAP envelope that wraps the "SOAP body" in the message.

- A SOAP header that describes how data is encoded.
- A SOAP body that contains application-specific messages.

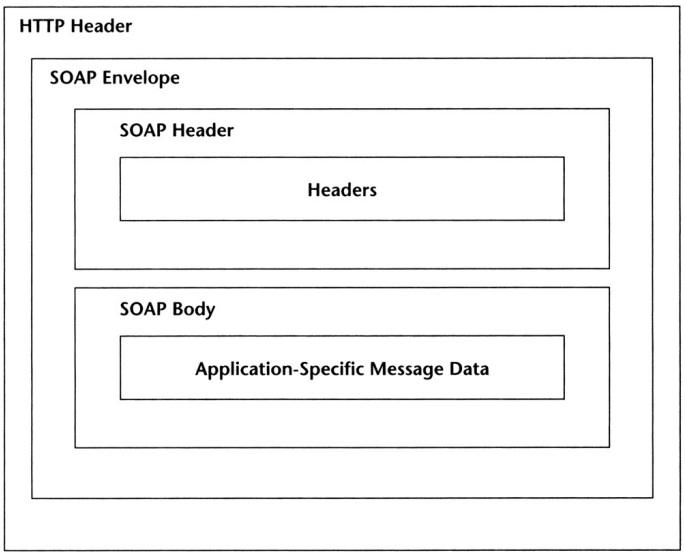


Figure 3.14: SOAP Layer basic Form

To demonstrate how SOAP works, suppose we have a simple SOAP based web service to query for a stock quote. A transaction between client and server is actually a HTTP POST and RESPONSE pair, where the request is like [41]:

```
GET /StockPrice HTTP/1.1
Host: example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
   xmlns:s="http://www.example.org/stock-service">
   <env:Body>
     <s:GetStockQuote>
          <s:TickerSymbol>IBM</s:TickerSymbol>
     </s:GetStockQuote>
   </env:Body>
</env:Envelope>
```

The response is a HTTP RESPONSE message like [41]:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
   xmlns:s="http://www.example.org/stock-service">
   <env:Body>
     <s:GetStockQuoteResponse>
          <s:StockPrice>45.25</s:StockPrice>
     </s:GetStockQuoteResponse>
   </env:Body>
</env:Envelope>
```

3.3.2 RESTful WS / ROA

Though SOA is a powerful architecture enjoying extensibility, neutrality and independence, it suffers from verbose XML format and increasingly complicated extension standards. In response to those issues, a more lightweight style of WS named RESTful web service have been proposed, which put emphasis on simple point-to-point message transmission over HTTP. The term "REST" is an abbreviation for **Representative State Transfer**, which comes from Roy Fielding's PHD dissertation "Architectural Styles and the Design of Network-based Software Architectures" [42]. According to the author, the name "Representative State Transfer" is intended to "evoke an image of how a well-designed Web application behaves: A network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use" [42].

RESTful web service has a simple linguistic architecture that can be summed up by four verbs (standard HTTP GET, POST, PUT and DELETE requests) and a set of nouns (resources provided by WS provider in the form of URLs). By taking advantage of the semantic of the HTTP protocol, all the services are naturally represented by a pair combination of verbs and nouns (See Table 5.1).

Noun \ Verb	GET	PUT	POST	DELETE
Collection URI, such as http://exp.com/res/	List the URIs and some other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URL is assigned automatically and is usually returned by the operation.	Delete the entire collection.
Element URI, such as http://exp.com/res/item	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	Replace the addressed member of the collection. If it doesn't exist, this will fail!	Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

Table 3.2: Verb & Noun Combination Table

To help people understand how RESTful web services are different from web services of other types, Leonard Richardson proposed a concept named **Richardson Maturity Model** at 2008 QCon [43], which demonstrates the steps that need to be taken before getting to a true RESTful web service. There are five levels in this model (See Figure 3.15):

- **Level 0: One resource (URI), one HTTP method (POST).** All of the services and functions are accessible by sending a HTTP POST request to the single URI. Typical SOAP based WS is a representative example of this level.

- **Level 1: Many resources (URIs), one HTTP method (POST).** In this level, the single big resource in the previous level is split down to many components and each is given a unique URI. This reduces the inner complexity of each resources and also cuts down verbose in communication messages.

- **Level 2: Many resources (URIs), each supporting multiple HTTP methods.** In this level, the HTTP protocol is no longer only used as a tunneling mechanism, but also plays an important role in semantic rules: By introducing the standard HTTP verbs (GET, POST, PUT, DELETE...) and their restrictions (GET requests have to be safe and idempotent, etc.), WS now has a uniform, self-explanatory interface that is compatible with the existing HTTP optimizing strategies (HTTP GET Caching, etc.) and the verbose in communication messages is cut down furthermore.

- **Level 3: Many resources (URIs) with hypermedia controls, each supporting multiple HTTP methods.** A level 2 WS will evolve to a true Restful WS if it adopts hypermedia controls in its response, which means the responding message to a consumer action should include the URIs of the

resources that could be used in the next operation. The point of hypermedia control is to introduce discoverability and make a WS self-explanatory.

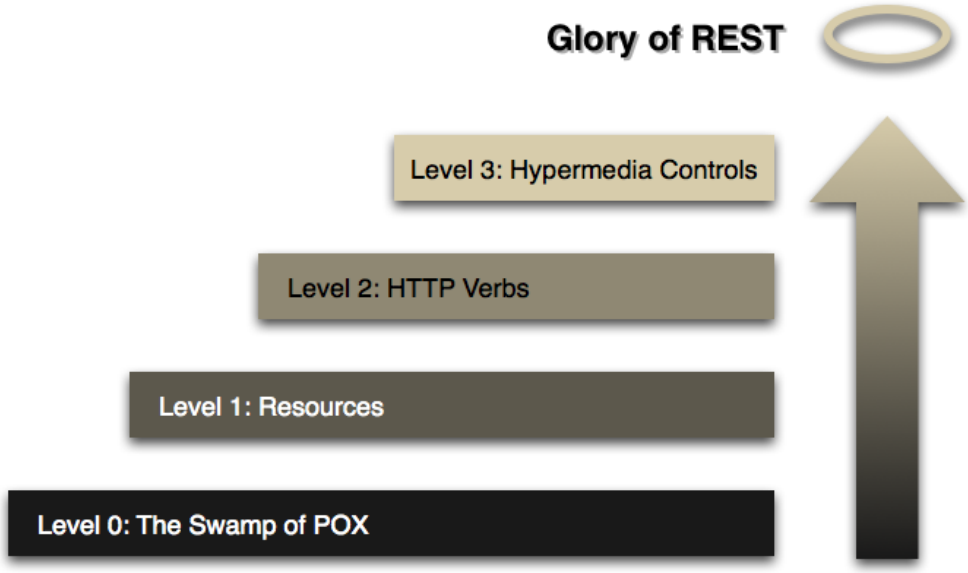


Figure 3.15: Richardson Maturity Model [44]

Let's revise the stock query example given in the last section to show how exactly RESTful web service works:

- In RESTful style the query message is just a HTTP GET request to a URL [41]:

```
GET /StockPrice/IBM HTTP/1.1
Host: example.org
Accept: text/xml
Accept-Charset: utf-8
```

- The response is just a normal HTTP response message containing data of stock price [41]:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<s:Quote xmlns:s="http://example.org/stock-service">
     <s:TickerSymbol>IBM</s:TickerSymbol>
     <s:StockPrice>45.25</s:StockPrice>
</s:Quote>
```

Unlike SOAP based web service, there are no verbose XML messages being transferred or parsed in the whole process. And each service transaction can be perfectly fit into a standard HTTP request & response pair. Thanks to the simplicity and conciseness, REST style has gained massive popularity on the public web since its invention. The implementations of REST include:

- The Atom Publishing Protocol. Which is an application-level protocol for publishing and editing web resources. [45]

- The Sun Cloud API. Which is a RESTful API for creating and managing cloud resources and can serve as a good example of resource media type documentation. [46]
- The Apache CouchDB. Which is a document-oriented database written in Erlang. It provides a RESTful JSON API, which can be accessed from any environment where HTTP requests are allowed. [47]

3.3.3 Comparison between SOAP and REST

The debate over the advantages of REST and SOA has never ended. Essentially they have very different design philosophy from the very beginning. The pros and cons of each style are summarized below:

- SOAP Pros:
  - Independent of languages, platforms, transport protocols
  - Great extensibility.
  - Functional interface-based communication scheme more suitable for distributed computing environments
- SOAP Cons:
  - More heavyweight, steep learning curve.
  - More verbose, requiring a special XML parser and causing heavier transferring payload.
  - More complicated because of those additional protocols invented by third-party vendors.
- REST Pros:
  - Language and platform independent.
  - Easy to integrate into existing HTTP web services.
  - Concise message with read/write semantic.
- REST Cons:
  - Assuming a point-to-point communication model, not suitable in a distributed computing environment.
  - Tied to HTTP protocol.

According to the comparison made by Pautasso etc. [48], RESTful WS is more suitable for "a tactical, ad hoc integration over the Web" and SOA WS is preferred in "a professional enterprise application integration scenarios with a longer lifespan and advanced QoS requirements" [48].

3.4 Push Technology

3.5 Summary

In the first half of this chapter, I have explored the technologies used by range camera-based motion sensors to generate useful information such as depth frame and skeleton. However, due to the limitation of bandwidth and computational capacity, it's still very expensive and unnecessary for many applications to directly transfer these data to the clients. This problem brings us back to previous two questions raised in the problem definition:

- **How to design an architecture that migrates computational intensive tasks from the clients side to server side?** This question is essentially equivalent to: Other than extracting and exposing three types of data (color, depth and skeleton) from the motion sensor, are there any additional tasks that can be done on the server side in order to reduce the computational burden on the client side? To find out the answer, let's take a look at the three motion sensor outputs again: The depth frame provides dense depth information of the scene and the skeleton data contains a set of coordinates of inferred body joint points. When those two kinds of data can be accessed simultaneously, it makes sense to use some joint coordinates to locate the regions of some critical body parts (such as hands and head) in the depth frame. In this way, further analysis can be specifically conducted on these regions to extract additional information such as hand gestures and facial expressions. All the analysis can be done totally on the server end and the clients only need to know the results. In this way, a large portion of the post processing is migrated from client side to server side, which makes a broader range of clients able to enjoy some high level functions that require significant amount of computational power.

- **How to design a data transfer scheme that is capable of transferring streams of data to clients using the HTTP protocol with good bandwidth consumption efficiency?** From the literatures reviewed, this problem can be tackled from two aspects:

  - First, to make clients able to receive high rate data streams, an efficient data transferring scheme needs to be carefully chosen. Considering the underlying protocol and targeted clients, I think Websocket can be used as the data streaming protocol. There are two main reasons for this choice: First, compared with the traditional server push approach, Websocket introduces less data overhead, therefore is more suitable for high rate updates scenario. Second, as a HTML5 standard feature, Websocket can be seamlessly incorporated into the underlying HTTP communication protocol used by my service without any custom plugins, which are required by some other HTTP based streaming technologies such as Adobe Flash.
  - Second, in many applications, what clients are concerned about is whether certain event happened or not rather than the frame sequence of the event. For example, a security alarm system based on

the motion sensor only needs to send warning messages to the clients when there is unauthorized entry rather than live streaming the situation at home without stopping. Thus it is reasonable to incorporate an event detection pipeline in my system, which can be accessed by clients through an event subscription and notification scheme. In this way, after the clients have subscribed to some events, no other data but some light-weighted notification message need to be sent to clients when the event is triggered. Such scheme can dramatically reduce the data transferring volume for those results-oriented clients.

In the second half of the chapter, I have reviewed literatures about some possible styles of web service which I might use for publishing the motion sensor functionalities. By comparing the advantages and disadvantages of different web service styles, I can now draw a conclusion that RESTful web service based on HTTP protocol is the most suitable one for my needs. There are several reasons supporting my conclusion:

- First and foremost, compatibility is an important factor regarding the diversity of mobile device platforms and standards. An HTTP based web service complying with REST standard is guaranteed to be accessible to most of the mobile devices using embedded web browsers with no or little specific modification.
- Secondly, unlike a distributed enterprise/B2B architecture where SOAP is preferable, the motion sensor web service is more likely to be organized in a tree structure, where different motion sensor servers rarely talk with each other but often need to be combined as a mashup so that data from multiple motion sensors can be accessed and synthesized together. So in terms of organization architecture, RESTful WS is more suitable.
- Last but not the least, as mentioned before, bandwidth consumption and data transferring volume are two critical factors that can't be ignored on mobile devices. Compared to SOAP/RPC, a well-designed RESTful web service has much less overhead in messages and can take the advantage of special techniques such as caching to cut down data traffic volume in advance. Therefore RESTful WS is more bandwidth-economical to the clients.

In summary, the literatures reviewed in this chapter give me some premier ideas to solve the challenges proposed in Chapter 2. However there are still some open problems left to be tackled in the architecture design and implementation part including:

- How to design a scalable architecture capable of handling a large number of high-rate streaming connections?
- How to design and implement the core data processing module in the framework to make it easy to extend?

Chapter 4

Architecture Design

To tackle the challenges proposed in chapter 2, the architecture of my web service framework is designed in a distributed way as shown in figure 5.1. It consists of one motion sensor server that physically linked to the motion sensor, and multiple distributed HTTP servers that are connected to the motion sensor server remotely via multiple TCP connections.

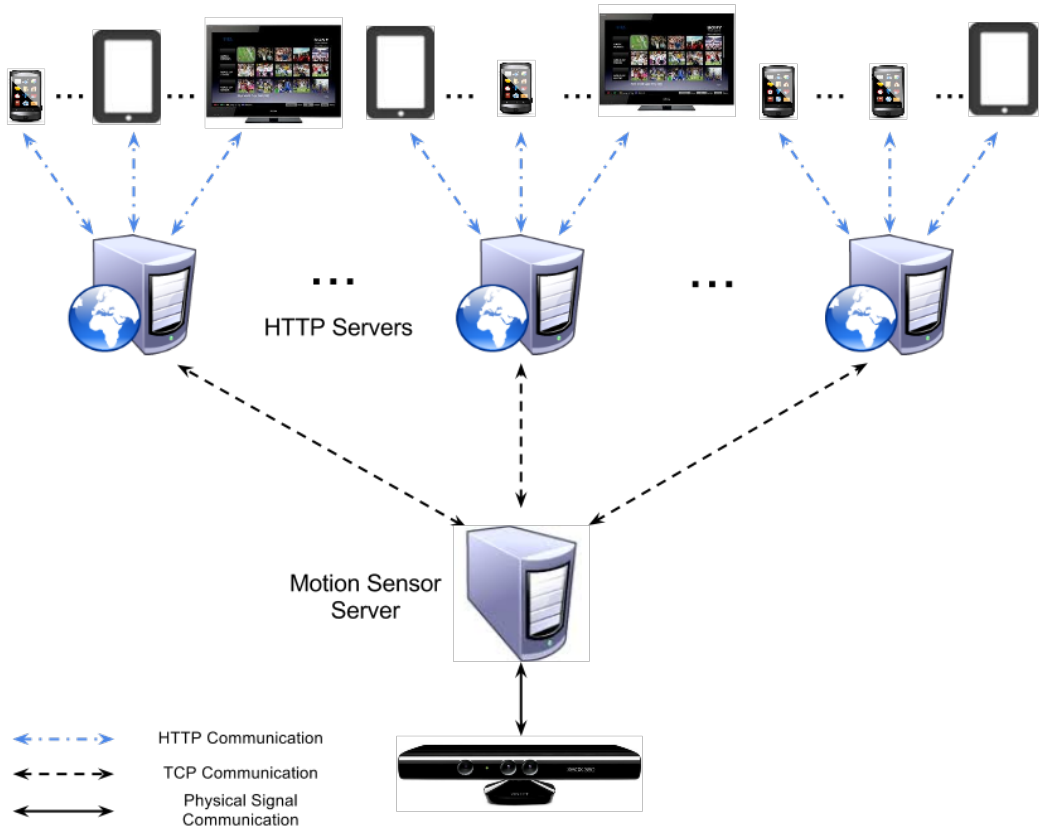


Figure 4.1: Overview of Web Service Architecture [7] [8] [9]

The architecture can be divided into four layers:

- An raw data accessing layer responsible for extracting raw outputs (color, depth, skeleton) from physically linked motion sensor.

- A gesture recognition layer that accepts the user's hand gesture definitions and conducts the hand gesture recognition.
- A interface Layer that integrates all the functionalities that the first two layers expose, exposes the data and functions in proper formats and sends them out to the upper layer through TCP connections.
- An HTTP Server Layer that retrieves data from middleware layer and exposes them as URLs to the clients in a RESTful approach through a pure HTML based interface.

The data accessing layer, the gesture recognition layer and the middleware layer all reside on the motion sensor server and the HTTP server layer is deployed on every HTTP server. The interaction relationship between each layer is shown in figure 4.2.

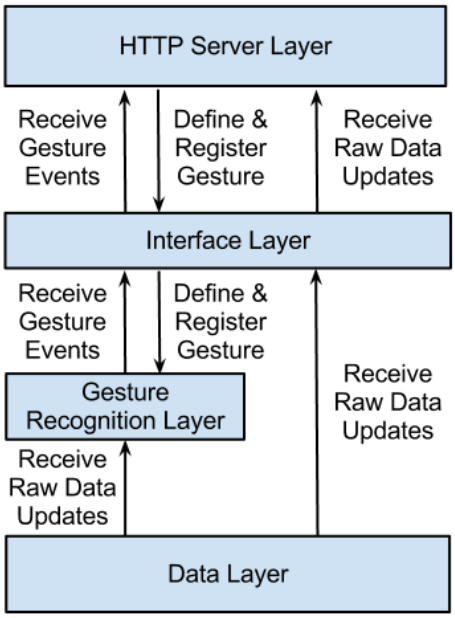


Figure 4.2: Interaction Relationship Between Layers

The rest part of this chapter will give an introduction about each layer's architecture from the bottom to the top.

4.1 Data Layer

The data layer is directly built upon the motion sensor programming API, which is normally provided by the motion sensor's manufacturer to enable developers to access and manipulate the data streams output by their device. As mentioned before, a typical motion sensor normally outputs three kinds of data, in which RGB and depth data are directly generated by related sub-sensors (color and depth sensor), and the posture

data (skeleton) will be extracted from the depth data afterwards by some build-in API functions using the technique introduced in Chapter 3 (See Figure 4.3).

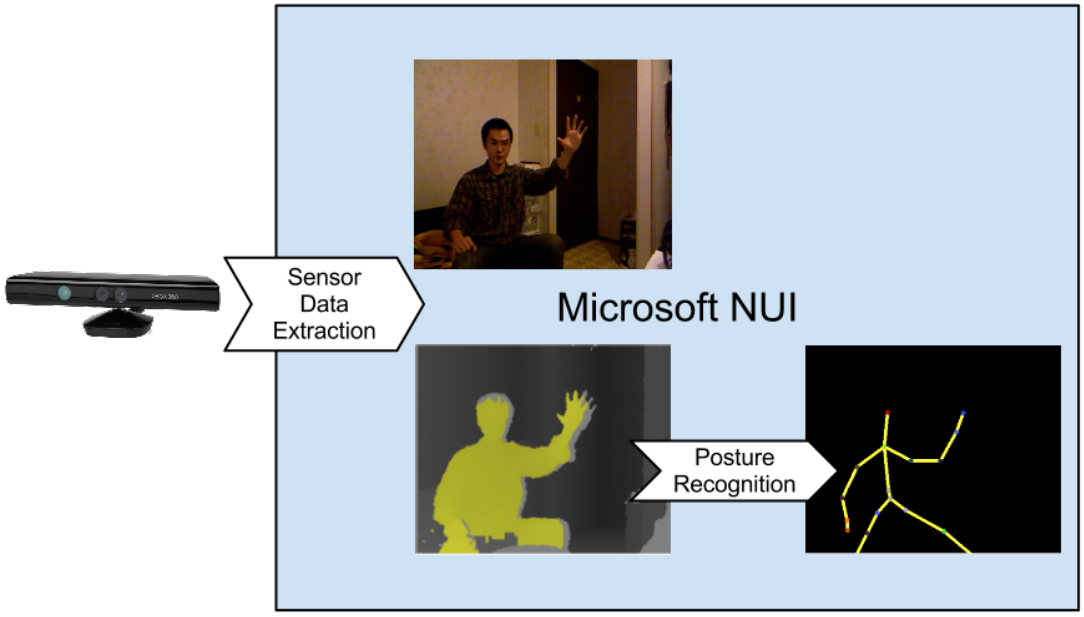


Figure 4.3: Motion Sensor Data Frame Generation Process

All types of data are normally updated at 30 times per second and provided in the form of streams. For easy extensibility and maintainability, my design of the data layer adopts an observer pattern, where each data stream is wrapped by a subject class to trigger a frame updating event when a new data frame is generated. Any upper layer component concerning about its updates just need to be implemented as an observer and registered into its observers queue. Whenever a frame updates take place, a notification thread which is constantly listening to updating events will capture the event and update every element in that queue one by one in a first-in-first-serve(FIFS) order (See Figure 4.4).

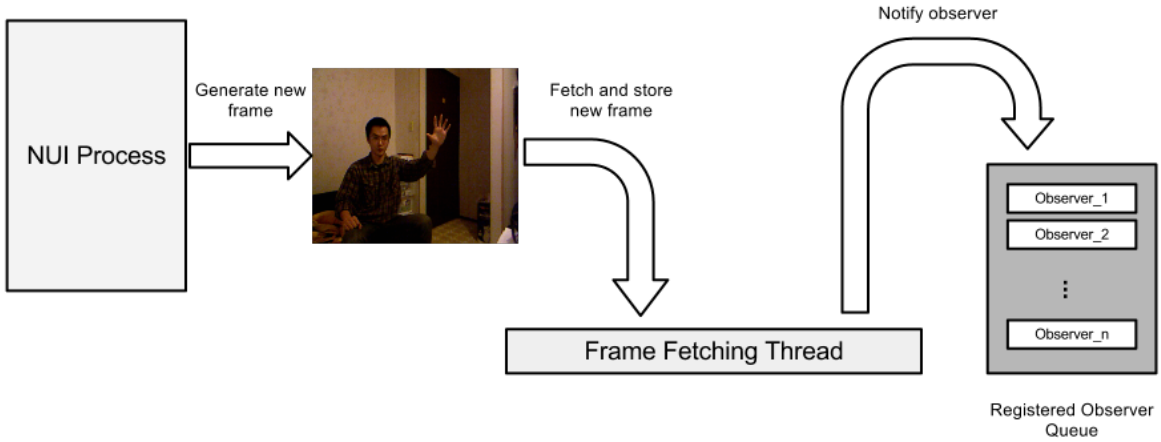


Figure 4.4: Data Accessing Process

A Unified Modeling Language(UML) class diagram of this layer is given in figure 4.5

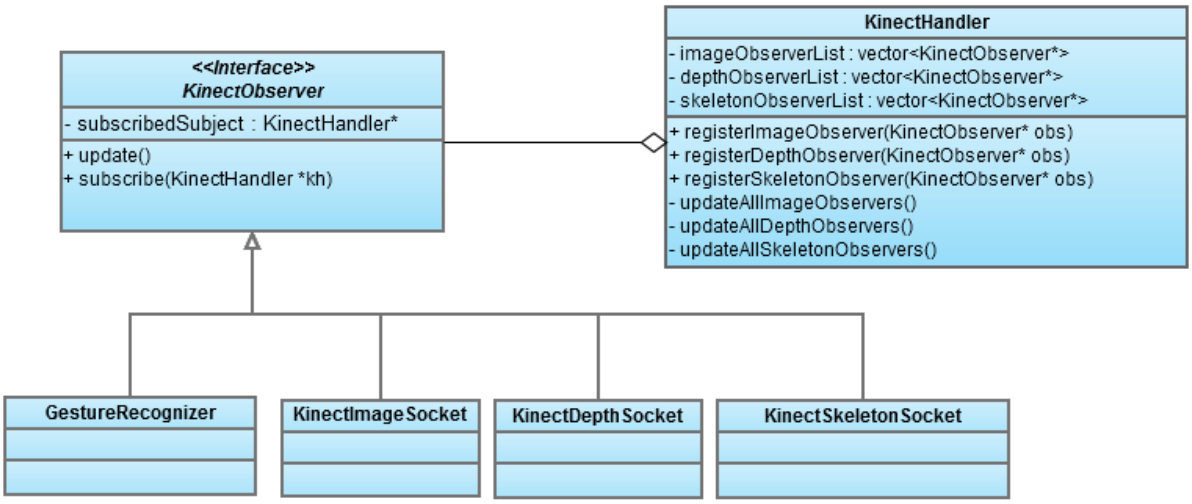


Figure 4.5: Data Accessing Layer UML Diagram

4.2 Gesture Recognition Layer

Freely defining and recognizing a set of body & hand gestures is an key functionality for many applications related to human computer interaction. While the build-in Skeleton abstraction of the human body provides a natural base to deal with body gestures, there is no function in the motion sensor API to conduct hand gesture analysis directly. For this reason, I have implemented a real-time hand gesture recognition pipeline, which consists of three major components:

- A hand analysis pipeline taking the output of the data accessing layer as input, through which an abstract hand model representing the current hand status will be generated from each depth frame at real-time.

- A gesture definition and recognition pipeline, which is able to robustly recognize gestures defined in the form of key frame sequences.

- A web-based event subscription component, which enables the user to define and subscribe to the gesture events they are interested in and receive notification once any subscribed event is triggered.

The gesture recognition layer plays a special role in the system: First, it needs access to both skeleton and depth data in data accessing layer, so it is implemented as a child class of ***MotionSensorObserver*** and registered to both ***SkeletonFrame*** and ***DepthFrame*** updates. Second, it serves as the gesture event generator in the system. Therefore it's also implemented as a subject class, so that any upper layer classes

interested in the gesture events can inherit the ***GestureObserver*** class and subscribe to the events by calling ***registerGestureObserver()*** (See Figure 4.6).

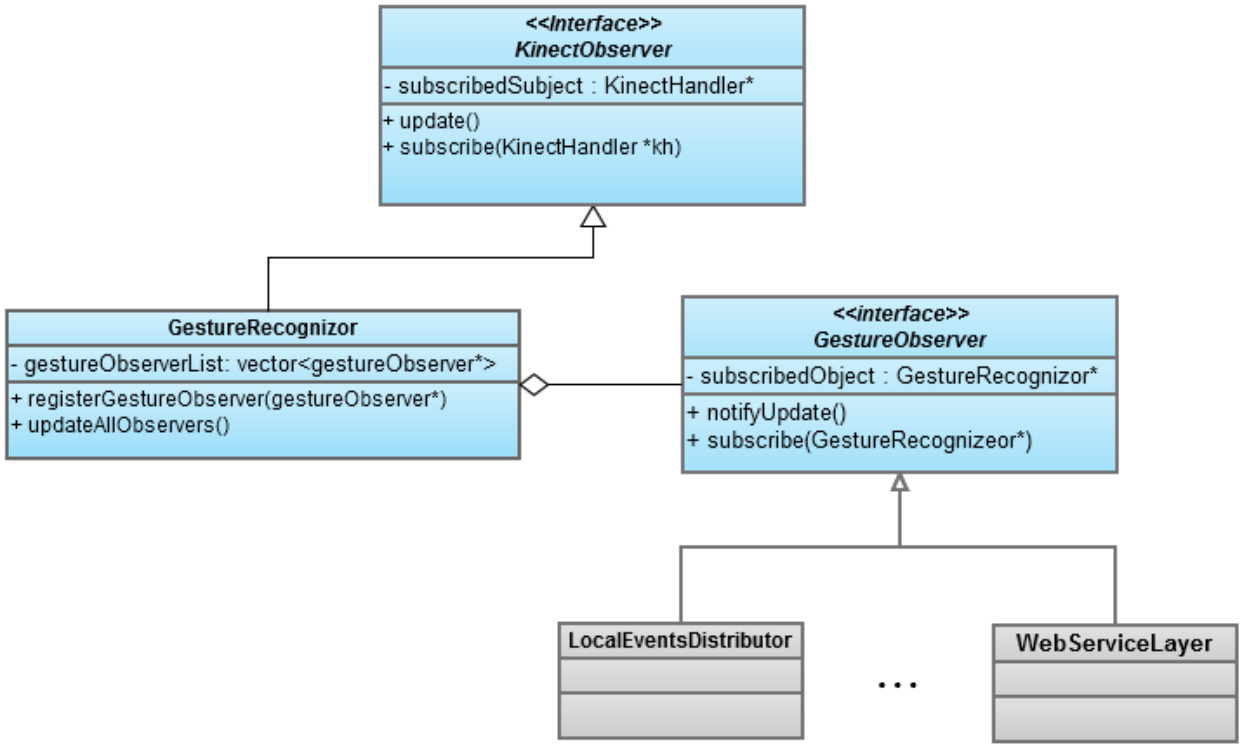


Figure 4.6: Gesture Recognition Layer UML Diagram

The ***GestureRecognizor*** maintains a map container ***gestureObserverMap*** that pairs each registered gesture with its event observers, where the gesture describer ***HandGestureAutomaton*** is the key and observer list ***vector<GestureObserver>*** is the value. Then in the ***update()*** method that is inherited from its parent class ***KinectObserver***, each gesture describer in the key set will be checked. If a specific gesture is detected, ***updateGestureObserver()*** will be called and all elements in the corresponding ***vector<GestureObserver>*** will be notified.

Consumers of ***GestureRecognizor*** can subscribe to any existing gestures or add self-defined gestures by invoking and passing its own reference along with the gesture's key frame sequence into ***registerGestureObserver(HandGesture, GestureObserver*)*** method, which will first check whether the parameter ***HandGesture*** is already in the key set of the ***gestureObserverMap*** or not. If so and the observer has never subscribed to the same gesture before, the observer's reference will be added to the corresponding observers list. Otherwise, a new gesture automaton will be generated by ***generateGestureAutomaton()*** method, paired with a newly-created observer queue and added as a new entry into the ***gestureObserverMap***.

4.3 Interface Layer

The responsibility of the interface layer is to integrate all the data and functionalities that the lower layers provide and enable the HTTP Server Layer to access them remotely. This layer consists of 4 discrete components (See Figure 4.7) including:

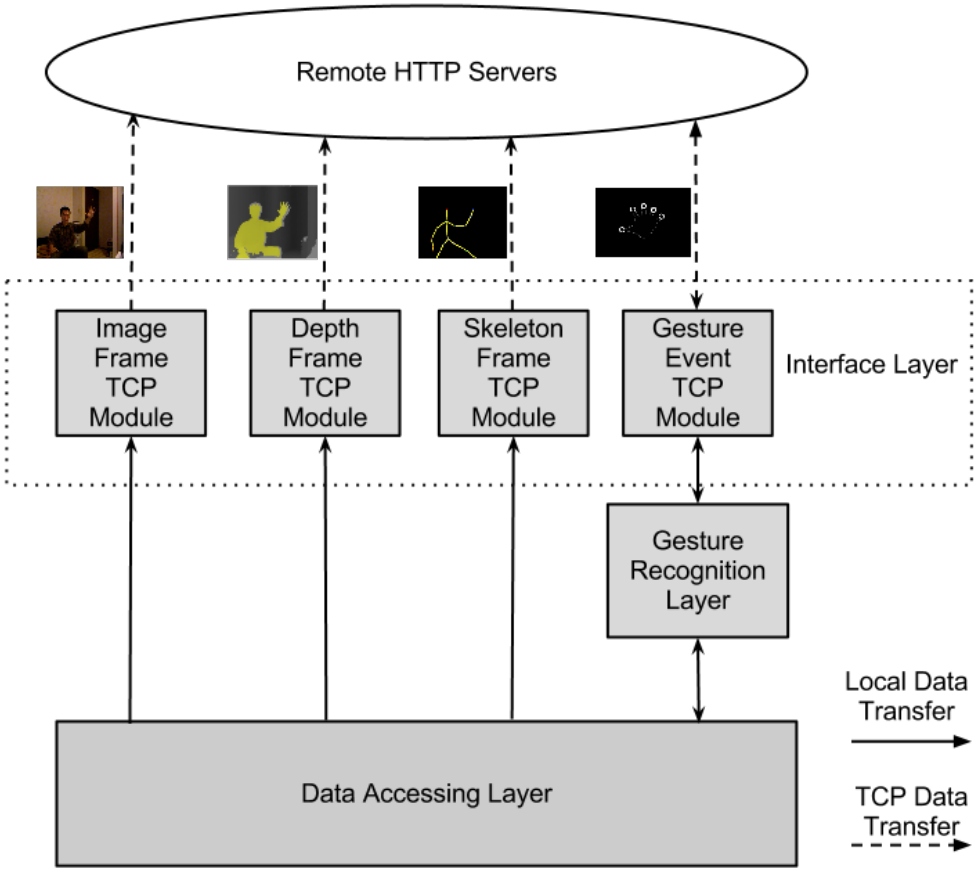


Figure 4.7: Interface Layer Architecture

- Color Frame TCP Module: Waiting for and maintaining TCP connections for RGB frames transmission.
- Depth Frame TCP Module: Waiting for and maintaining TCP connections for depth frames transmission.
- Skeleton Frame TCP Module: Waiting for and maintaining TCP connections for skeleton frames transmission.
- Gesture Event TCP Module: Waiting for and maintaining TCP connections for gesture events transmission.

Each module is implemented as an observer to the related data source, which will retrieve and deal with the updated frames or events in its ***update()*** method. A complete UML diagram of each components is given in figure 4.8.

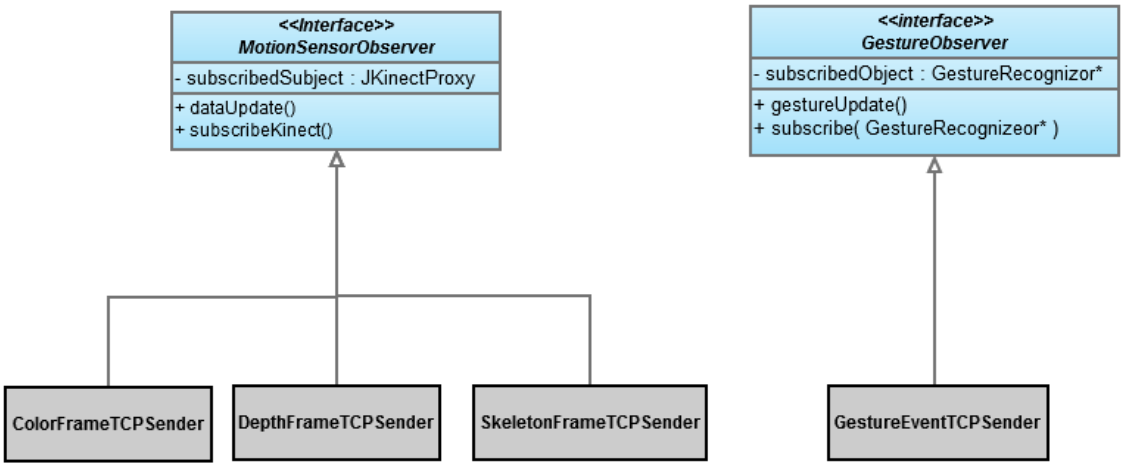


Figure 4.8: Interface Layer UML Class Diagram

4.4 HTTP Server Layer

The HTTP server layer consists of multiple remote HTTP servers (See Figure 4.9) and each server maintains four TCP connections with the motion sensor server to retrieve four data streams (RGB, depth, skeleton, gesture event) in real time.

On every HTTP server, there is a TCP module responsible to receive data frames and events from the motion sensor server through TCP connections. The received data will be stored as local resources and mapped to different URLs, which are accessible to mobile clients through HTTP transactions (See Figure 4.10). The HTTP Server layer serves as the interface that finally exposes all the data and functionalities provided by the lower layers to mobile devices. To make my service usable to a wide range of mobile devices, my design of this layer strictly complies with the following principles:

- Communicating over HTTP protocol using messages in standard HTML/XML format.
- Resource Oriented Architecture, every functionality exposed is taken as a resource and mapped to a URL.
- Each resource is represented by a pure HTML/JavaScript based web page complying with HTML5 standard, no third-party/propriety plugin is needed.
- Only standard HTTP requests are used to interact with URLs, and their usage strictly conforms the principles recommended by W3C. For example, GET is a safe method to retrieve resources without

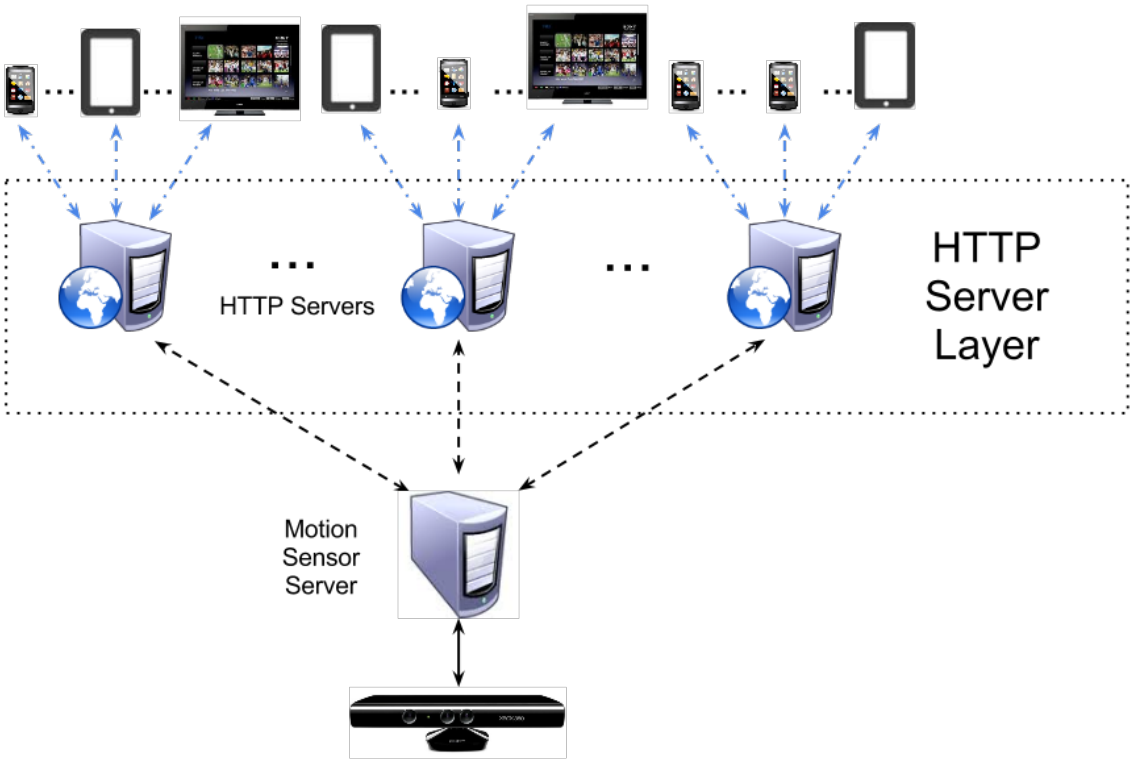


Figure 4.9: HTTP Server Layer Overview [7] [8] [9]

introducing any changes on the server end, while POST and DELETE are used to change resources on the server, etc.

There are four kinds of resources accessible to the clients in our system: RGB frames, depth frames, skeleton frames and gesture events. For each resource, my system provides two access models: getting it as snapshot or getting it as stream. Those two models are mapped to two different URLs like http://KinectServer/Kinect1/DepthFrame/snapshot and http://KinectServer/Kinect1/DepthFrame/stream. When the server has received a request for the snapshot URL, it will simply return a HTML page containing the current frame to the client. When the request is made to the stream URL, the server will return a HTML page containing javascript code to establish a Websocket connection between the client and the HTTP server. After clients executed the code and got the websocket connection established, data frames will be constantly pushed to clients in the form of stream until the connection is terminated (See Figure 4.11).

All URLs in the HTTP server layer are organized in an tree structure (See Figure 4.12), where each node represents a URL that is formed by linking together all the strings along the direct path from the root node to itself. If the URL client visit is a leaf node (.../Snapshot or .../Stream), the server will respond with the requested data in a correct form (snapshot or stream). Otherwise, the server will return the client with a HTML page containing the URLs of its parent and all of its children. In this way, the web service is self

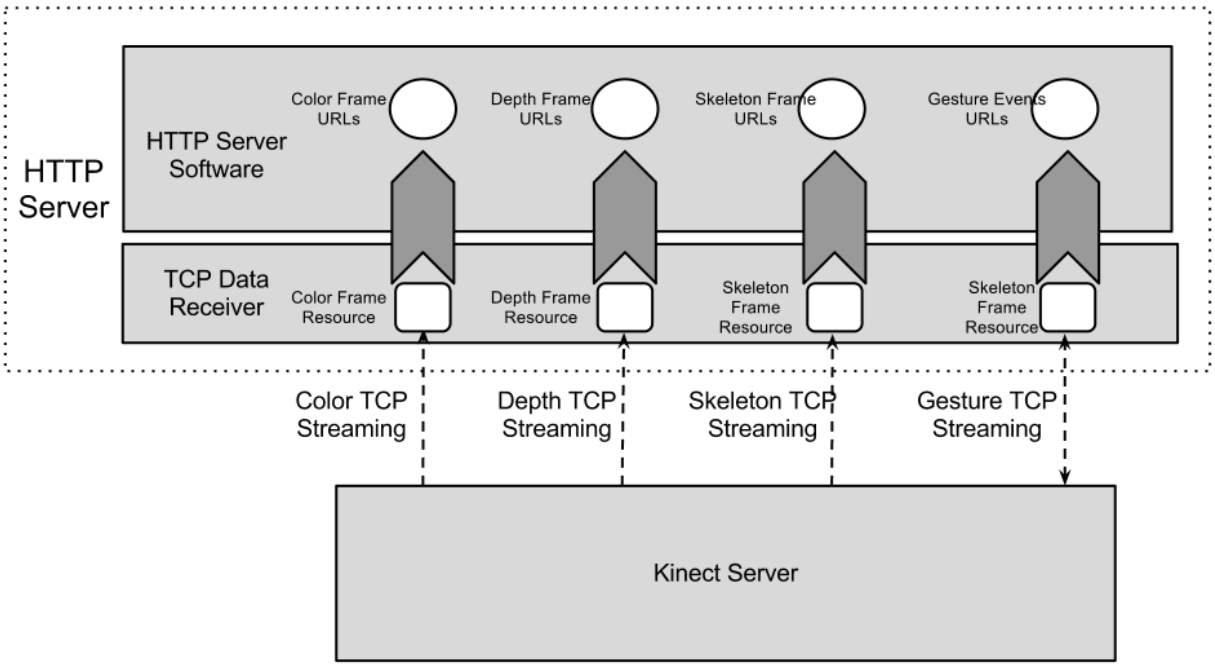


Figure 4.10: Architecture of Each HTTP Server

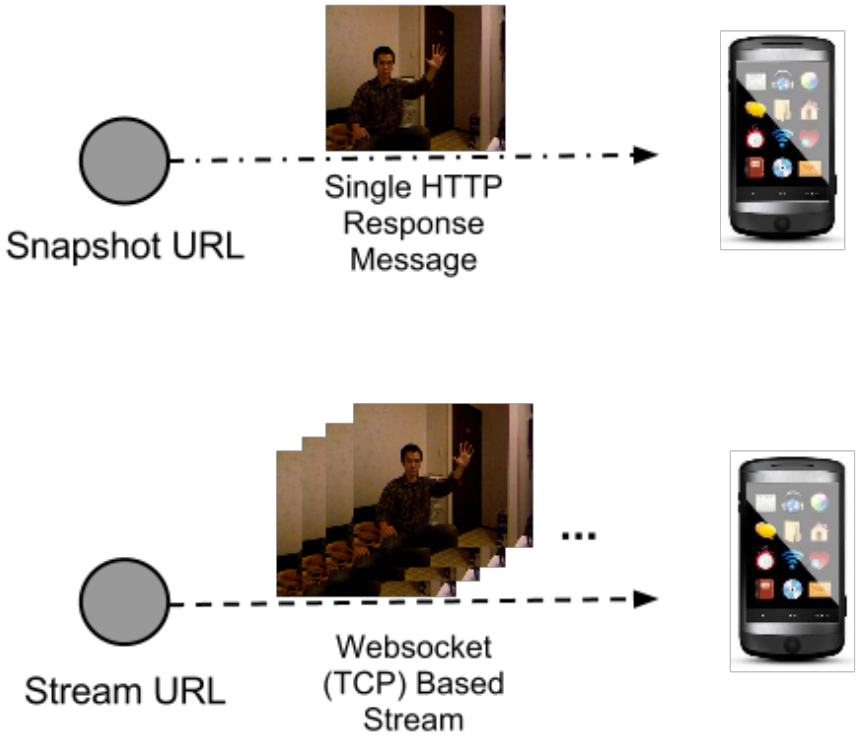


Figure 4.11: Two Accessing Models [8]

explanatory: The user can always find the resource he or she wants by navigating through the hyperlinks in the page without pre-knowledge about the service's structure.

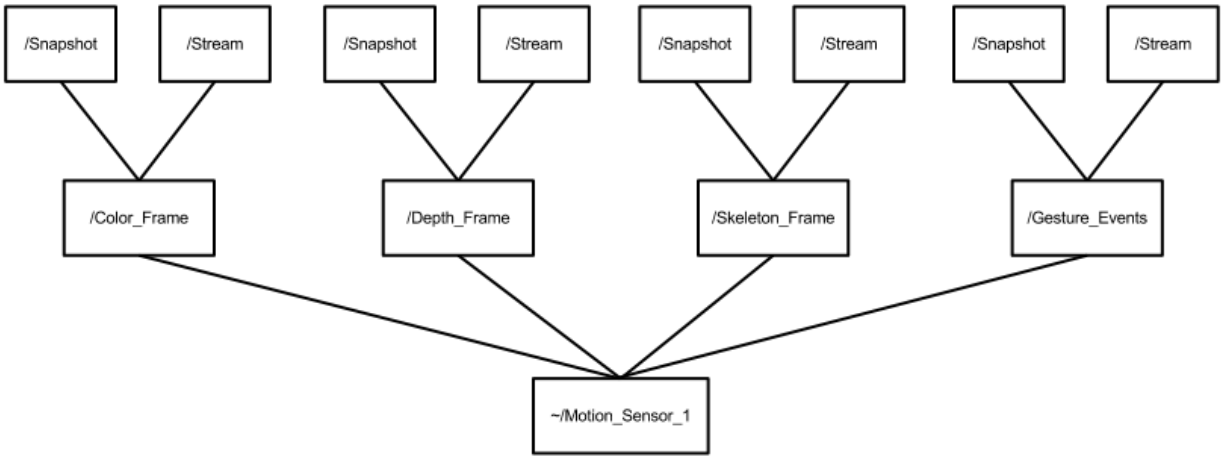


Figure 4.12: URLs Organization

4.5 Summary

In this chapter, I proposed a web service architecture to solve those problems raised in chapter 2. Solution to each problem is summarized below:

- **Client End Computational Power Consumption**
  - By adding a gesture recognition pipeline, all the computational-expensive processing tasks are transfered to the server end. All the data exposed by my web service (color, depth, skeleton frames and gesture events) can be directly used in some applications by clients with limited computational capacity.
- **Client End Bandwidth Consumption**
  - By using Websocket to transfer the data frames updated at high rates, the web service can deliver streams of data with minimal overhead to the clients.
  - By introducing an event subscription and notification scheme, the data that need to be transfered to the result-oriented clients can be further cut down significantly.
- **Scalability**
  - By introducing a separate HTTP server layer on the top of the motion sensor server, the service's serving capacity is multiplied and the architecture becomes scalable: The pressure of serving large number of clients simultaneously is transferred from the motion sensor server to those HTTP

servers, which are optimized for such task. Moreover the service providers can easily adjust the number of HTTP servers used in this layer according to the actual workload, which give them additional flexibility.

- **Extensibility**
  - By designing the data accessing layer in an observer pattern, adding extensions to the existing functionalities is very easy: Additional function modules just need to be implemented as an observer class and subscript to the related data updates. It's not necessary to change the existing code base.

Chapter 5

Experiment & Evaluation

To evaluate the performance of my architecture design, I have implemented a motion sensor web service based on Microsoft Kinect sensor. Details about my implementation is given in Appendix A. In this chapter, I will conduct three experiments on the prototype to evaluate the proposed solutions to three of the problems stated in chapter 2:

Experiment Goal	Experiment Description
To evaluate the client computational burden.	**Client Compatibility Evaluation.** By trying to access the web service from different browsers on different platforms, this experiment intends to evaluate whether the computational capacity requirement on client side is lowered to a degree that could be met by most of current mobile devices.
To evaluate the scalability.	**Scalability Evaluation.** This experiment is designed to give a estimation about the maximum number of clients that the system could serve simultaneously.
To evaluate the bandwidth consumption.	**Data Push Experiment.** This experiment is going to evaluate the system's data transmission performance in terms of bandwidth consumption and latency under both low and high transmission rates.

Table 5.1: Experiment Goals

5.1 Client Compatibility Experiment

All the functionalities provided by the motion sensor web service is designed to be accessible by web browsers on mobile devices, which generally have very limited computational capacity. Unlike PC clients where a great variety of web browser options are available, mobile clients usually have very limited choice of web browsers. As some of the new technologies my framework relies on are still under development (such as websockets), different versions/kinds of web browser may implement the same technology according to different, sometimes incompatible, versions of drafts, which may bring about some compatibility issues. In this experiment, we

are going to evaluate how well our web service can be accessed directly through a web browser on different mobile platforms.

5.1.1 Experimental Setup

- Kinect Server: A HP workstation with 2 Intel Xeon 3.20GHz processors 4GBs RAM and connected to the network via 100 Mbps Ethernet. A Kinect is physically linked with this machine via USB and driven by the self-developed Kinect Server application running under 64-bit Windows 7 Enterprise SP1.
- HTTP Server: A Lenovo desktop with an Intel Core i5-2400 processor 8GBs RAM and connected to the network via 100 Mbps Ethernet. A self-developed middleware is deployed on this machine and linked to the Kinect Server via TCP connections. Apache Tomcat 7.0 is running under running 64-bit Windows 7 Enterprise SP1 to host resources and serve HTTP requests. I tested all the services the system provides including:
  - Single framed color/depth/skeleton data.
  - Streamed color/depth/skeleton data.
  - Gesture/posture events subscription & notification.
- Client Side Devices: A desktop running 64-bit Windows 7 Enterprise SP1, tested web browsers include:
  - Windows Internet Explorer9 (version 9.0.8112.16421)
  - Google Chrome (version 19.0.1084.52)
  - Mozilla FireFox (version 12.0)
  - Opera (version 11.64)

  A Mac desktop running MacOS X 10.7, tested web browsers include:
  - Safari (version 5.1)
  - Google Chrome (version 19.0.1084.52)
  - Mozilla FireFox (version 12.0)
  - Opera (version 11.64)

  An iPad2 running iOS 5.1.1, tested web browsers include:
  - Safari for iOS (version 5.1)
  - Opera Mini (version 7.0.2)

  An Acer Iconia A500 Tablet running Android 4.0, tested browsers include:
  - Default Android Web Browser

- Google Chrome for Android Beta (version: 0.18.4409.2396)
- Mozilla Firefox (version: 10.0.4)
- Opera Mobile (version 12.0.3)

5.1.2 Process, Results and Analysis

For each client device, I used each of the browsers listed under its category to access each of the services to see whether it's accessible or not. The results for each platform is given below:

PC	Chrome	Firefox	Opera	IE9
RGB Static Link	pass	pass	pass	pass
Depth Static Link	pass	pass	pass	pass
Skeleton Static Link	pass	pass	pass	pass
RGB Stream	pass	pass	fail	fail
Depth Stream	pass	pass	fail	fail
Skeleton Stream	pass	pass	fail	fail
Events Push	pass	pass	fail	fail

Table 5.2: Compatibility Test: PC

Mac	Chrome	Firefox	Opera	Safari
RGB Static Link	pass	pass	pass	pass
Depth Static Link	pass	pass	pass	pass
Skeleton Static Link	pass	pass	pass	pass
RGB Stream	pass	pass	fail	fail
Depth Stream	pass	pass	fail	fail
Skeleton Stream	pass	pass	fail	fail
Events Push	pass	pass	fail	fail

Table 5.3: Compatibility Test: Mac

The results show that the service works particularly well with the Chrome and Firefox, which passed all the tests regardless of the platform. While the Opera fails all the streaming/pushing tests due to lack of Websocket support. What worth noticing is the Safari browser, which passed the streaming/pushing tests on iOS but failed on MacOS. This is because the Safari used on the Mac is at version 5.1.7, which is older than the one used by the iPad (version 6.0) and conforms to an expired protocol of Websocket that is not compatible with current one. From this experiment, we can conclude that, on each platform, there are at least two browsers supporting all the services, which means the full functionalities of the web service can be directly accessible via web browser on most major desktop/mobile platforms.

Android	Chrome	Firefox	Opera
RGB Static Link	pass	pass	pass
Depth Static Link	pass	pass	pass
Skeleton Static Link	pass	pass	pass
RGB Stream	pass	pass	fail
Depth Stream	pass	pass	fail
Skeleton Stream	pass	pass	fail
Events Push	pass	pass	fail

Table 5.4: Compatibility Test: Android

iOS	Chrome	Opera	Safari
RGB Static Link	pass	pass	pass
Depth Static Link	pass	pass	pass
Skeleton Static Link	pass	pass	pass
RGB Stream	pass	fail	pass
Depth Stream	pass	fail	pass
Skeleton Stream	pass	fail	pass
Events Push	pass	fail	pass

Table 5.5: Compatibility Test: iOS

5.2 Scalability Experiment

This experiment is designed to give an estimation about the maximum number of mobile clients that our system can serve simultaneously. As described in Chapter 4, the system is designed to be deployed in a distributed way, where a motion sensor server is physically linked with Kinect, and several distributed HTTP servers each maintaining four TCP connections with the motion sensor server to retrieve data streams. When a mobile client comes in, the client establishes an HTTP conversation with one of those HTTP servers to access the services (See Figure 5.1).

In such an architecture, the maximum number of mobile clients is determined by:

$$M_s \times M_h \tag{5.1}$$

where M_s denotes maximum number of concurrent TCP connection threads that can be maintained on the Kinect server, and M_h denotes maximum number of simultaneous HTTP requests that each HTTP Server is capable to handle. As M_h varies in different kinds of HTTP servers, which is largely determined by the server's software implementation and hardware speed, this experiment is focused on evaluating M_s measured by following method: As all data frames updates at a constant rate, in order to keep this refreshing rate

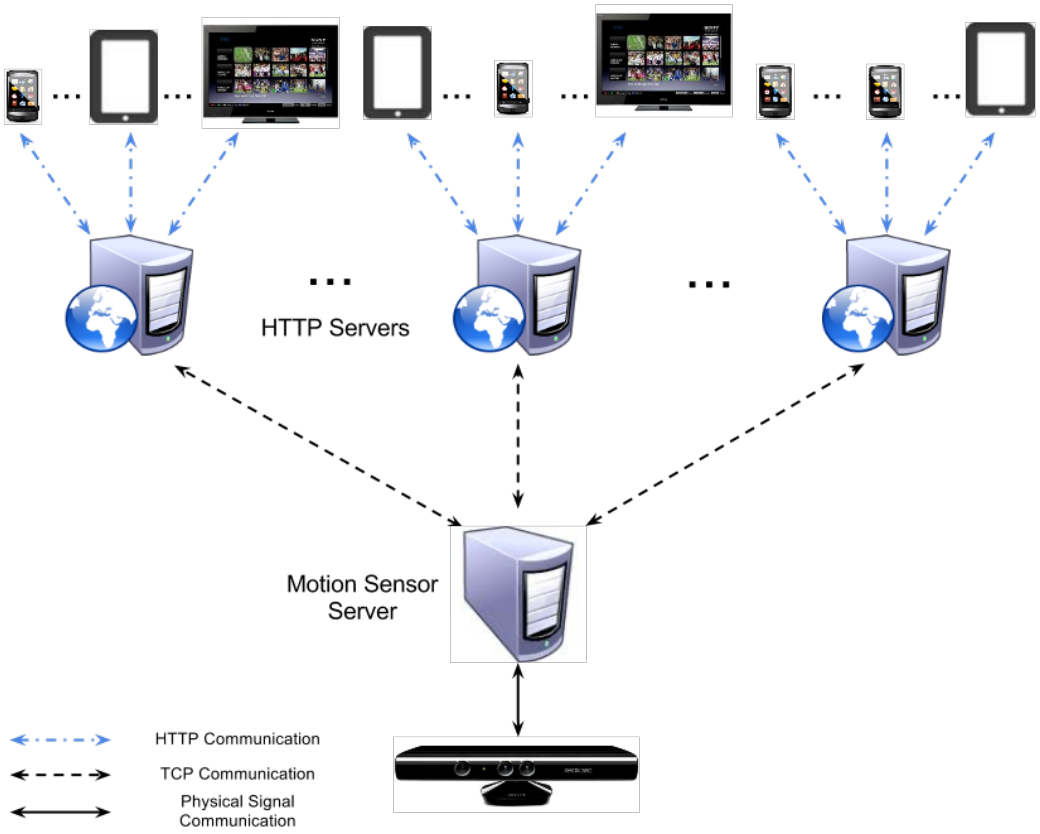


Figure 5.1: Distributed Architecture [7] [8] [9]

without delay, all working threads on the motion sensor server need to finish processing current frame before the next update. M_h is determined by measuring the maximum number of TCP connections the Kinect server could handle while still be able to finish all the data processing work on time.

5.2.1 Experimental Setup

- Kinect Server: A HP workstation with 2 Intel Xeon 3.20GHz processors 4GBs RAM running 64-bit Windows 7 Enterprise SP1 and connected to the network via 100 Mbps Ethernet. A Kinect is physically linked with this machine via USB and driven by our self-developed Kinect Server application. To simulate the workload for the gesture recognition pipeline in typical scenario, four 3-keyframes gestures are registered for recognition.

- HTTP Server: A Lenovo desktop with an Intel Core i5-2400 processor 8GBs RAM running 64-bit Windows 7 Enterprise SP1 and connected to the network via 100 Mbps Ethernet. Our self-developed middleware is deployed on this machine. In order to simulate n HTTP servers, n instance of the middleware is initiated on $3 * n$ different ports connected to the Kinect server via n different TCP

connections.

- Client Side Devices: None.

5.2.2 Process, Results and Analysis

This is an experiment with multiple iterations. Each iteration is a five-minute long session, during which I measured the average time taken by each working thread to process one data frame and find out the maximum t_{max}. If t_{max} is shorter than the time interval t_f between two successive data frame updates, which means on average every thread can finish processing current frame before the arrival of the next one, I will increase the number of TCP connection n by one and run another iteration until n reaches a number n_m that makes the t_{max} longer than t_f. In this case, n_m is the threshold to prevent some thread's processing rate from catching up with the frame updating rate. So M_h, the maximum number of TCP connections the Kinect server could handle without affecting its performance, is $n_m - 1$.

When the skeleton detection function is disabled, the frame rate for depth/skeleton stream is $30 fps$ so $t_f = 1/30 \approx 33ms$. In this case, the t_{max}/n relationship is given in figure 5.2. The result shows when the number of connections is less equal than three, the frame output rate can keep up with the frame generation rate. When the connection number rises to above three, the time of sending one frame will be longer than the time of generating one frame – if such status lasts long enough, data congestion will happen. So we can conclude that when the skeleton detection function is disabled, the maximum number of HTTP servers that could be linked to the Kinect server is three.

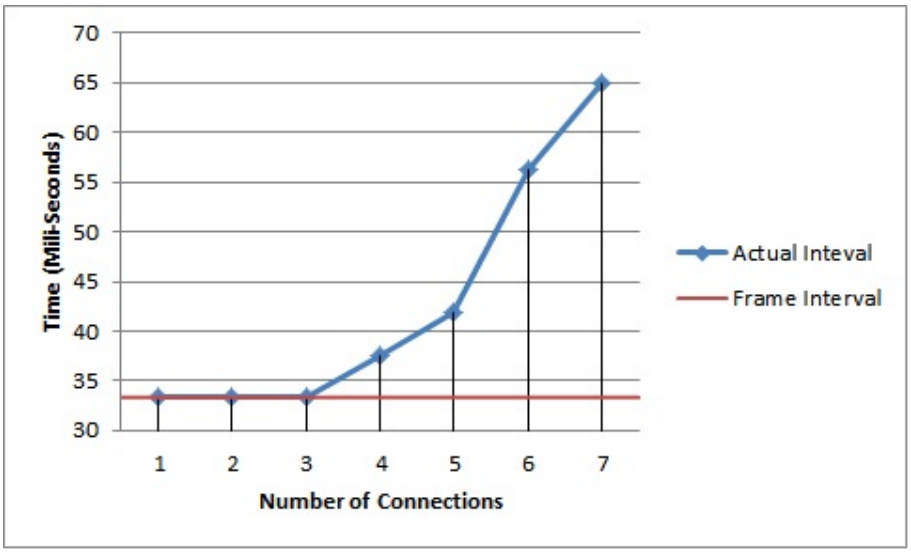


Figure 5.2: Scalability Test: No Skeleton. The red line indicates the default frame interval of Kinect when the skeleton detection is disabled (33ms). The blue line shows the actual frame interval that changes along with the number of connections. The result shows when the connection increases to above 3, the data transferring module will begin to lag down the system's performance.

When the skeleton detection function is disabled, the frame generation interval for depth/skeleton stream

will be increased to $72ms$, which will give the data transferring module more time to transfer data. My experiment shows the frame transferring time will start to exceed the frame generation time when the number of connections reaches seven (See Figure 5.3), which means in this case the maximum number of HTTP Servers is six.

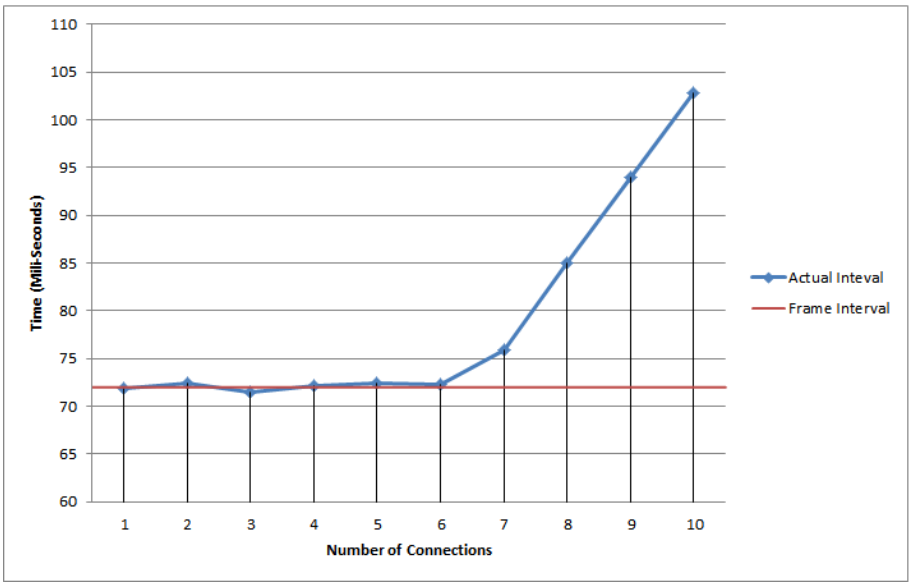


Figure 5.3: Scalability Test: With Skeleton. The red line indicates the default frame interval of Kinect when the skeleton detection is enabled (72ms). The blue line shows the actual frame interval that changes along with the number of connections. The result shows when the connection increases to above 6, the data transferring module will begin to lag down the system's performance.

Given the max number of HTTP servers that could be connected to the motion sensor server, We can now get an estimation about the total serving capacity of this framework by measuring the serving capacity of each HTTP server. Suppose the maximum number of connections that could be processed simultaneously by a HTTP server is around 256, which is a typical number [49], then the serving capacity is around 768 $(256 * 3)$ when the skeleton is disabled and around 1536 $(256 * 6)$ when the skeleton is enabled.

5.3 Data Push Experiment

5.3.1 Low Rate Event Push Experiment

Long distance event registration and notification is one of the most common application scenarios of my system. When a server side event (registered gesture or posture) triggered, the registered clients need to be notified. In a low rate event rate scenario where events take place at most once or twice per second, there are two approaches to get the job done: Client side polling and server side pushing. Client side polling means it's the client's responsibility to check the event took place or not. As the client is completely blinded about the server side, it needs to initiate check requests repetitively at a constant rate. Server push means it's

the server's responsibility to inform the registered clients when some event happens, what the client needs to do is listening to the corresponding port for incoming messages, and send feedbacks after receiving event notifications (see Figure 5.4).

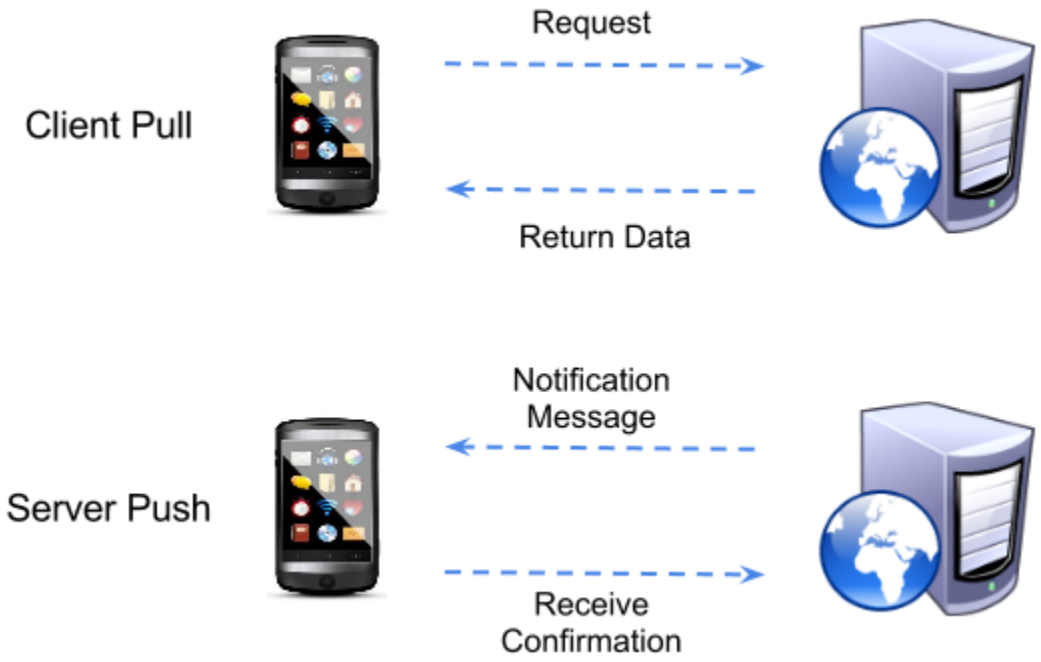


Figure 5.4: Pull and Push [8]

In this experiment, I'm going to compare the performance of websocket based event push and standard HTTP polling in terms of bandwidth consumption and update latency.

Experiment Setup

- Kinect Server: A HP workstation with 2 Intel Xeon 3.20GHz processors 4GBs RAM and connected to the network via 100 Mbps Ethernet. A Kinect is physically linked with this machine via USB and driven by our self-developed Kinect Server application running under 64-bit Windows 7 Enterprise SP1. The gesture detection pipeline in our system is enabled with one gesture registered. We will use a predefined sequence of frames, in which the registered gesture performed 20 times, as input to the detection pipeline to simulate a fixed series of gesture events.

- HTTP Server: A Lenovo desktop with an Intel Core i5-2400 processor 8GBs RAM and connected to the network via 100 Mbps Ethernet. Our self-developed middleware is deployed on this machine running under 64-bit Windows 7 Enterprise SP1.

- Client Side Devices: A Nexus 7 tablet accessing our service from an IP outside of our server's local area network. The web browser used is Google Chrome 18.0.

Process, Result and Analysis

In this experiment, I played a predefined gesture event sequence (which contains 50 events in five minutes) on the Kinect server for twice: The first time was for the pull test, where the client kept checking the event, whose status is representing as a resource, via HTTP request every one second. The second time was for the push test, where the client was registered on the HTTP server, and listened to incoming notifications from the Websocket. For each iteration, I measured the following data:

- Volume of transferred data: The total amount of data uploaded and downloaded during the test.
- Update latency: The time between an event being generated on the server and being received by the mobile device.

The statistics of this experiment is given below:

Interaction Method	Data Transferred Per Time		Data Transferred Per Event	
	Uploaded	Downloaded	Uploaded (averaged)	Downloaded (averaged)
Polling	385 bytes	103 bytes	2310 bytes	618 bytes
Pushing	0 bytes	3 bytes	0 bytes	3 bytes

Table 5.6: Low Rate Push Experiment

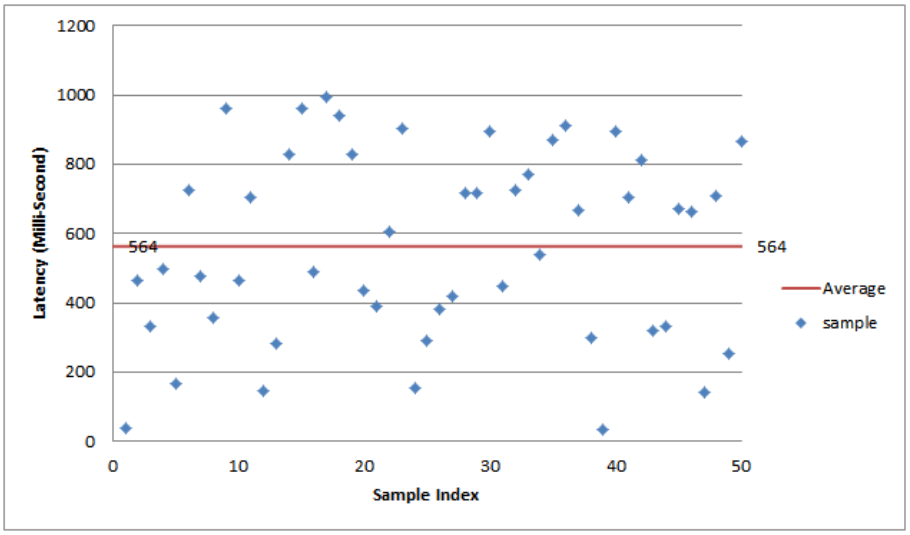


Figure 5.5: The Latency of HTTP Poll. The blue dots show the actual latency for each event. The red line indicates the average latency per event, which is 564ms.

As expected, the websocket pushing method achieves significant performance gain comparing to the HTTP polling:

- The average latency for the HTTP polling is 564ms, which is close to its theoretical value $pollingInterval/2 = 1000ms/2 = 500ms$, while the average latency for the websocket pushing is $12.33ms$ which is largely

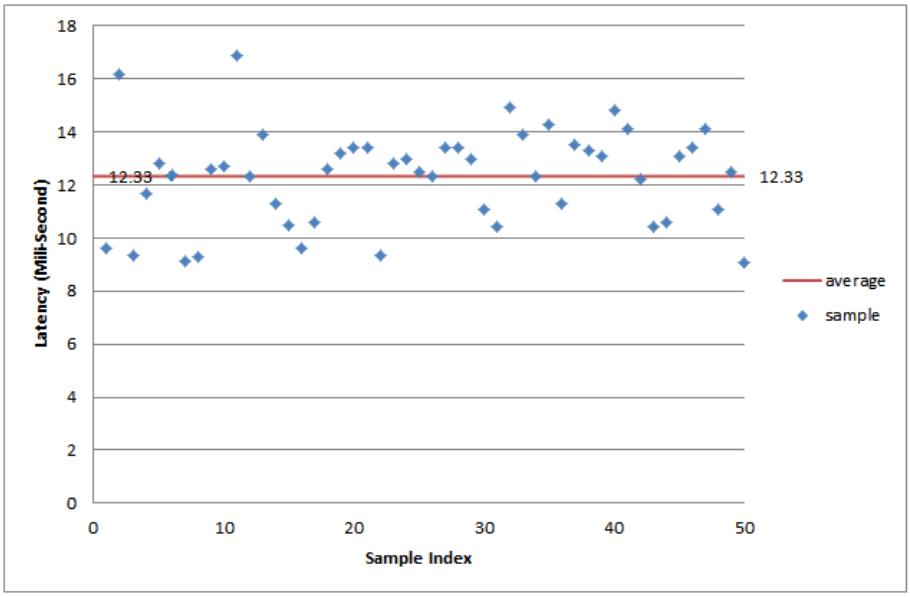


Figure 5.6: The Latency of Websocket Push. The blue dots show the actual latency for each event. The red line indicates the average latency per event, which is 12.33ms.

determined by the **Round Trip Time (RTT)** between the server and client.

- The average data transferred per event using HTTP polling is 2928 bytes (2310 bytes uploaded + 618 bytes downloaded), which is depending on the event rate (As the polling rate is constant, the lower the event rate, the more checks needs to be made for each event, therefore the more bytes need to be transferred). On comparison the data transferred for each event using websocket pushing is constantly 3 bytes.

5.3.2 High Rate Data Push Experiment

In cases where the client application takes the web service as an input for real time interactions (for example, mobile games), data frames need to be received by clients at very high rate in order to guarantee responsive enough user experience. Such a high data rate makes great demands on bandwidth and connection latency, as a result, Websocket streaming becomes the only option to transfer data. In this experiment, I used a self developed mobile game that controlled by skeleton stream to evaluate the performance of websocket data push.

Experiment Setup

- Kinect Server: A HP workstation with 2 Intel Xeon 3.20GHz processors 4GBs RAM and connected to the network via 100 Mbps Ethernet. A Kinect is physically linked with this machine via USB and driven by our self-developed Kinect Server application running under 64-bit Windows 7 Enterprise SP1.

- HTTP Server: A Lenovo desktop with an Intel Core i5-2400 processor 8GBs RAM and connected to the network via 100 Mbps Ethernet. Our self-developed middleware is deployed on this machine running under 64-bit Windows 7 Enterprise SP1.

- Client Side Devices: An iPad connected to network via 100 Mbps Ethernet with an IP inside of our servers' local area network, which is placed in the same room with the Kinect Server and the HTTP Server. The self-developed game running on it communicates with the HTTP Server via Websocket.

A overview of the experiment scenario is shown in figure 5.7.

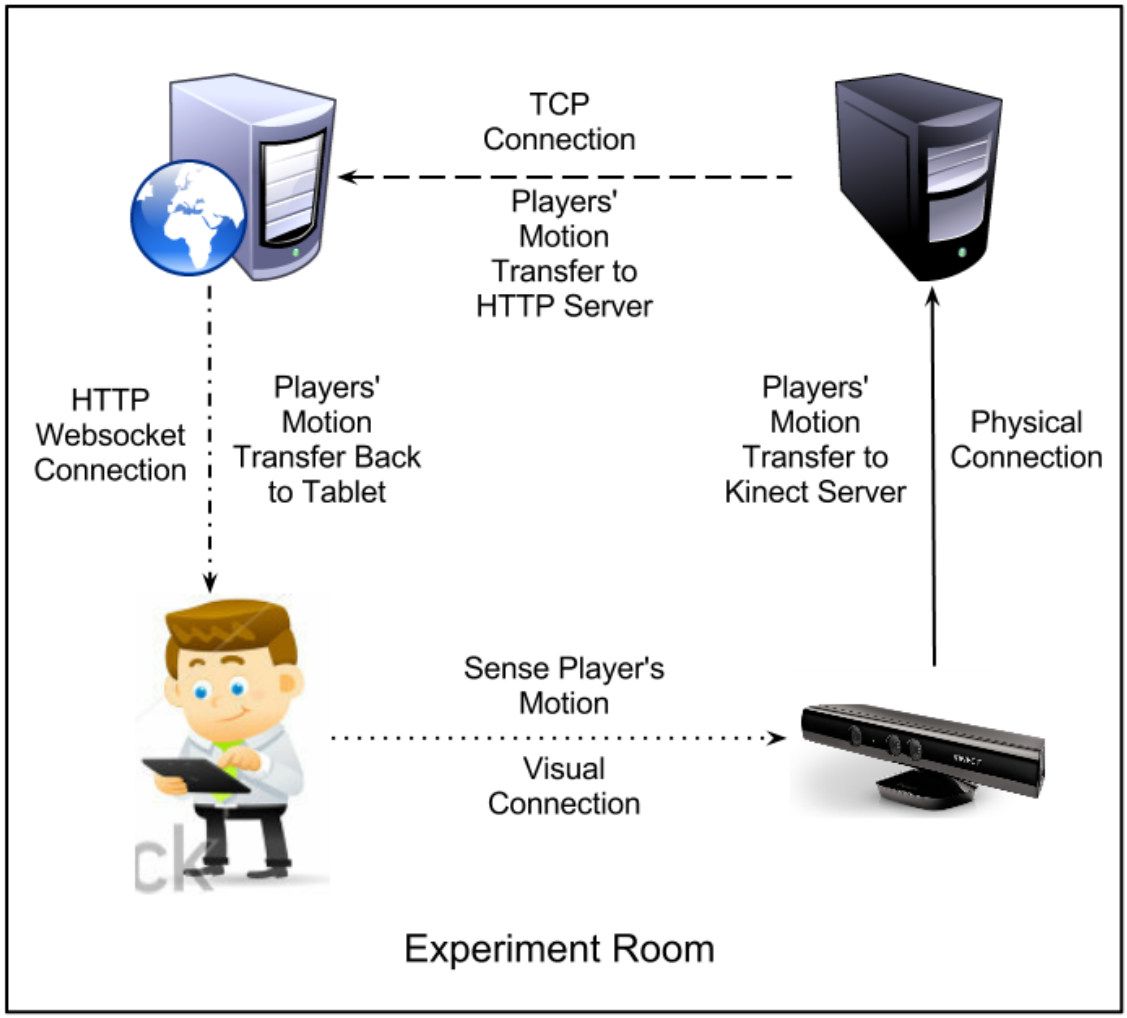


Figure 5.7: Real Time Application Experiment [1]

Process, Result and Analysis

The client application is a first-person-shooter like game that retrieves the skeleton stream via websocket in real time and uses it to determine the player's point of view/model movement. Therefore the player can

walk, jump, crouch and make any other movements in the game world just like what he/she does in real life. During the 30 minutes experiment, I collected following data:

- Average bandwidth consumption, which will be used to determine the bandwidth necessary to ensure a smooth stream transfer without frame loss or congestion.
- Average latency, which is the average time from a skeleton frame being updated on the server to it being reflected in the game.

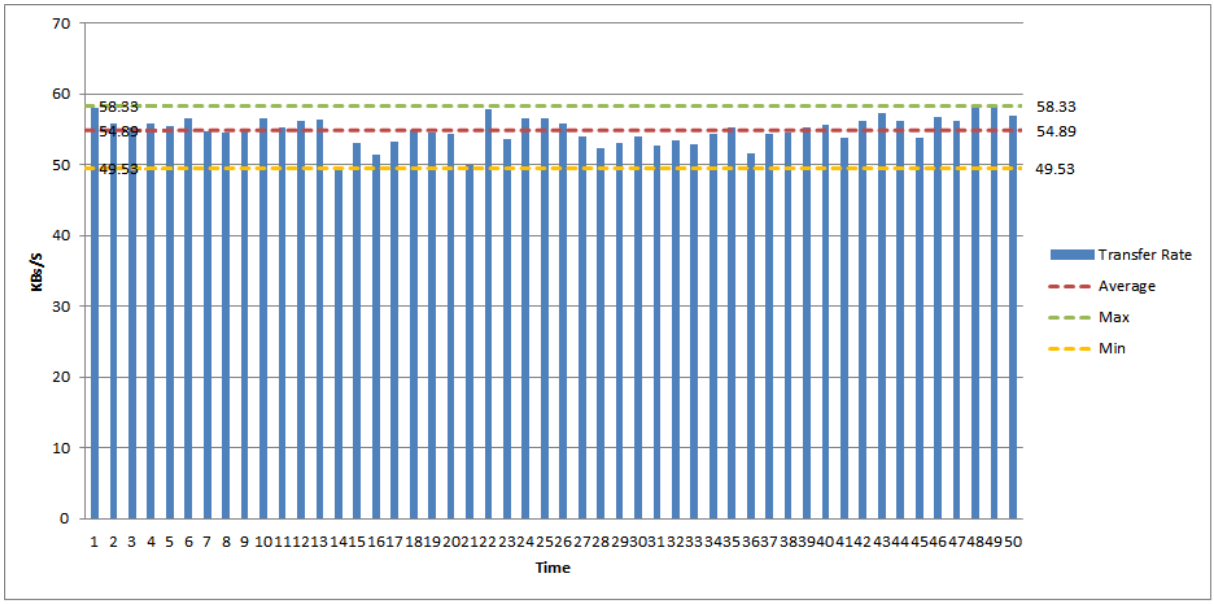


Figure 5.8: Bandwidth Consumption. This graph shows the bandwidth consumption during the experiment. A measurement was made every 36 seconds and there were totally 50 measurements in the 30 minutes experiment session (shown as blue bars). The average bandwidth consumption as indicated by the red dash line is 54.89 KB/s, and the peak bandwidth consumption as indicated by the green dash line is 58.33 KB/s.

The experiment shows the average bandwidth consumption during the 30 minutes experimental session is 54.89 KB/s, and peaks at 58.33 KB/s (See Figure 5.8), which means the bandwidth requirement for accessing the skeleton stream is around 60 KB/s – a figure that can be easily met by most mobile devices connected through WIFI or 3G network. The average latency is 14.11 ms (See Figure 5.9), that means even in a game runs at 60 fps whose frame interval is 1000/60 = 16.67ms, the player can expect to see his/her current movement to appear in the next frame – which will guarantee a very responsive experience.

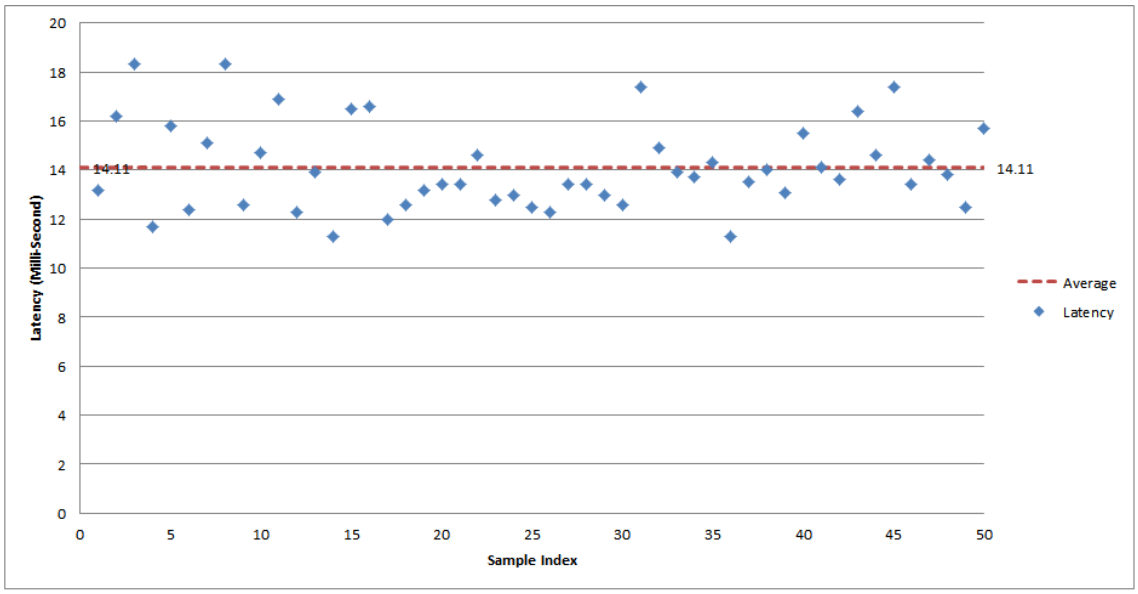


Figure 5.9: The Latency of Skeleton Stream This graph shows the skeleton transferring latency during the experiment. A measurement was made every 36 seconds and there were totally 50 measurements in the 30 minutes experiment session (shown as blue dots). The red dash line indicates the average latency per measurement, which is 14.11ms.

5.4 Summary

In this chapter, I have conducted three experiments to evaluate the framework's performance in three aspects, and the result about each of them is summarized as below:

- **Compatibility.** The first compatibility experiment shows that the snapshot version of all contents (RGB, depth, skeleton and gesture event) can be accessed from all tested browsers on any of the tested platforms (Windows, MacOS, Android and iOS). And on each of the tested platforms, there is at least one browser that could access the stream version of these contents. Put it in another way, the full functionality of the framework can be accessed on all tested platforms.

- **Scalability.** In the scalability experiment, I have testified that the maximum number of HTTP servers that could be linked to Kinect server without causing data congestion is three (when skeleton function is disabled) and six (when skeleton function is enabled). The serving capacity of the framework is derived to be 768 (when skeleton function is disabled) and 1536 (when skeleton function is enabled) assuming the average serving capacity of those HTTP severs is 256.

- **Data Transferring Performance.** In the low rate data push experiment, I have made a comparison between HTTP polling and websocket pushing. The result shows websocket pushing has a remarkable performance advantage over HTTP polling in terms of bandwidth consumption (3 bytes per event to 2928 bytes per event) and latency (12.3 ms to 564 ms). In the high rate data push experiment, the websocket based data transferring scheme exhibits that it is fast enough to be used in real time

interactive applications (14.11 ms latency) and bandwidth economical enough for most mobile devices (60 KB/s bandwidth requirement).

Chapter 6

Summary & Contribution

In this research, I proposed a framework to publish motion sensor's functionalities as a web service that accessible from a broad range of clients. This framework is proven to achieve following goals:

- Platform independence. By adopting a pure HTTP protocol based transferring scheme and a pure HTML/Javascript based interface, the full functionality of the web service can be accessed from any web browser that conforms to the HTML5 standard regardless of the underlying OS and hardware.
- High scalability. By using a distributed architecture, the framework has the potential to deal with 700+ consistently connected (via websocket) clients simultaneously under high rate motion sensor data transferring scenarios. And the architecture design also makes it possible to flexibly adjust the serving capacity according to the actual workload.
- High data transferring efficiency. The websocket based data streaming scheme exhibits good compatibility (supported by at least one web browser on each of four major platforms) and is proven to be highly efficient – it lowers the latency and bandwidth consumption to a degree that makes the motion sensor data streams could be easily used by mobile devices for real-time interaction.
- Low computational burden on client side. A server side data processing module is embedded in the framework to conduct hand gesture recognition on behalf of the clients. And its architecture is designed in a way that could be easily extended. Therefore the framework has the potential to transfer any additional computational intensive works from the client side to the server side to make high level functionalities of motion sensor accessible by mobile clients with limited computational capacity.

In terms of application, the framework mainly contributes from two aspects:

- Firstly, it's the first framework to enable mobile devices to access motion sensor functionalities remotely, which breaks down the barrier and brings the application of motion sensor into a much broader area. Some example applications based on this framework include:
  - Mobile games use motion sensor as a controller. Games can retrieve player's skeleton in real time just as the game used in the experiment. This is particularly interesting for a mobile FPS game, as the player can move along with the mobile device's screen, it will give him/her an unique virtual environment experience.

  - Home monitoring/security system. The user can have a motion sensor monitoring certain place worth concern (living room for example) and register his/her mobile device to some special events (such as skeleton appearing) through the framework. Then the user can get real time notification about those events (some one entered his/her living room) even when he/she is thousands miles away.

- Secondly the restful architecture makes it easy to combine multiple such motion sensor web services as one mashup. In regarding to this, it's the first framework that allows clients to access arbitrary numbers of motion sensors simultaneously. This is very important in many research fields such as 3D reconstruction, where data from multiple sensors needs to be obtained and synthesized at the same time.

Chapter 7

Future Works

7.1 Improve Websocket Compatibility

The current websocket server used by my framework prototype is an implementation of the protocol described by W3C SOAP Version 1.2 [50] and only compatible with client applications conforming to the same protocol, which greatly limits client's browser options – excluding popular ones such as Opera and older versions of Safari, Chrome and Firefox. A solution to this problem is to redesign the current server to make it compatible with multiple websocket protocol versions at the same time. This can be achieved by introducing a protocol branching scheme: Whenever a connection request arrives, the server will first discriminate the version of protocol that the client browser uses (which can be obtained from the request header), then it will branch into the corresponding protocol and response with the correct message format accordingly.

7.2 Apply Temporal Compression Techniques

The current data transferring module only applies spatial image compression (raw image to jpeg) and completely ignores the information redundancy between adjoining frames. As the RGB, depth and skeleton streams are all smoothly transformed sequences, adjacent frames usually share high degree of similarity. Temporal compression techniques will take advantage of this feature: Instead of sending an entire frame each time, it only sends the differences between the current frame and the previous one if they are similar. In this way, a properly selected temporal compression method could further cut down the current bandwidth consumption by a large degree.

7.3 Advanced Server Side Image Analysis

The current hand gesture pipeline is just a primary showcase of server side image processing. Actually the server can do much more than that given the RGB, depth and skeleton information. Following are several possibilities:

- A more delicate hand model could be extracted from depth image, which is able to capture the figure tips' motion in 3D space. Such a hand model could significantly enrich the gesture set and improve the

recognition accuracy.

- User's face information can be captured in a similar way as hands. After that further information such as facial expression can be extracted based on it.
- Based on the skeleton and the depth information of the scene, high level activity recognition can be conducted. It's possible to make the motion sensor server know whether the people in its view are walking, running or chatting with others. So the framework can respond with proper actions depending on what the framework is used for.

Bibliography

[1] L. Hackwith, "using the kinect in a wheelchair." http://www.levihackwith.com/using-the-kinect-in-a-wheelchair/g, January 2011.

[2] M. CSAIL, "Mit kinect based interface." http://www.ros.org/wiki/mit-ros-pkg/KinectDemos, April 2012.

[3] D. Bonvin, "Connecting kinects for group surveillance." http://actu.epfl.ch/news/connecting-kinects-for-group-surveillance/, December 2010.

[4] D. ZAX, "Can kinect help detect autism?." http://www.technologyreview.com/view/427909/can-kinect-help-detect-autism/, May 2012.

[5] D. Corporation, "Openni." www.openni.org, 2011.

[6] OpenKinect, "Openkinect." http://openkinect.org, March 2012.

[7] J. Carter, "Sony bravia internet video review." http://www.techradar.com/reviews/pc-mac/software/operating-systems/sony-bravia-internet-video-696498/review?src=rss&attr=all, June 2010.

[8] PSD, "Psd mobile phone, black cellphone icon download psd file." http://www.psdpsds.com/psd-files/28/psd-mobile-phone-black-cellphone-icon-download-psd-file.html, June 2012.

[9] S. Grever, "Samsung galaxy tab 10.1 review - the best android tablet?." http://www.psdpsds.com/psd-files/28/psd-mobile-phone-black-cellphone-icon-download-psd-file.html, August 2011.

[10] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, WSNA '02, (New York, NY, USA), pp. 88–97, ACM, 2002.

[11] M. Ghanem, Y. Guo, J. Hassard, M. Osmond, and M. Richards, "Sensor grids for air pollution monitoring," in *In Proc. 3rd UK e-Science All Hands Meeting*, 2004.

[12] R. Cardell-oliver, K. Smettem, M. Kranz, and K. Mayer, "Field testing a wireless sensor network for reactive environmental monitoring," in *In International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pp. 14–17, 2004.

[13] C. R. Baker, K. Armijo, S. Belka, M. Benhabib, V. Bhargava, N. Burkhart, A. D. Minassians, G. Dervisoglu, L. Gutnik, M. B. Haick, C. Ho, M. Koplow, J. Mangold, S. Robinson, M. Rosa, M. Schwartz, C. Sims, H. Stoffregen, A. Waterbury, E. S. Leland, T. Pering, and P. K. Wright, "Wireless sensor networks for home health care," in *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 02*, AINAW '07, (Washington, DC, USA), pp. 832–837, IEEE Computer Society, 2007.

[14] F. Bernardini and H. Rushmeier, "The 3d model acquisition pipeline," *Computer Graphics Forum*, vol. 21, no. 2, pp. 149–172, 2002.

[15] B. Curless and S. Seitz, "3d photography." SIGGRAPH course note accessed from:`http://www.primesense.com/en/technology/115-the-primesense-3d-sensing-solution`, 1999.

[16] J. Hecht, "Photonic frontiers: Gesture recognition: Lasers bring gesture recognition to the home." `http://www.laserfocusworld.com/articles/2011/01/lasers-bring-gesture-recognition-to-the-home.html`, January 2011.

[17] I. Albitar, P. Graebling, and C. Doignon, "Robust structured light coding for 3d reconstruction," in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pp. 1 –6, October 2007.

[18] M. Young, E. Beeson, J. Davis, S. Rusinkiewicz, and R. Ramamoorthi, "Viewpoint-coded structured light," in *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pp. 1 –8, June 2007.

[19] P. Inc, "the Primesense 3D sensing solution." `http://www.primesense.com/en/technology/115-the-primesense-3d-sensing-solution`, November 2011.

[20] T. B. Moeslund, A. Hilton, and V. Krüger, "A survey of advances in vision-based human motion capture and analysis," *Comput. Vis. Image Underst.*, vol. 104, pp. 90–126, November 2006.

[21] R. Poppe, "Vision-based human motion analysis: An overview," *Comput. Vis. Image Underst.*, vol. 108, pp. 4–18, October 2007.

[22] D. Grest, J. Woetzel, and R. Koch, "Nonlinear body pose estimation from depth images.," in *DAGM-Symposium'05*, pp. 285–292, 2005.

[23] D. Anguelov, B. Taskarf, V. Chatalbashev, D. Koller, D. Gupta, G. Heitz, and A. Ng, "Discriminative learning of markov random fields for segmentation of 3d scan data," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 2, pp. 169 – 176 vol. 2, June 2005.

[24] Y. Zhu and K. Fujimura, "Constrained optimization for human pose estimation from depth sequences," in *Proceedings of the 8th Asian conference on Computer vision - Volume Part I*, ACCV'07, (Berlin, Heidelberg), pp. 408–418, Springer-Verlag, 2007.

[25] M. Siddiqui and G. Medioni, "Human pose estimation from a single view point, real-time range sensor," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pp. 1 –8, June 2010.

[26] E. Kalogerakis, A. Hertzmann, and K. Singh, "Learning 3d mesh segmentation and labeling," *ACM Trans. Graph.*, vol. 29, pp. 102:1–102:12, July 2010.

[27] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake, "Real-Time Human Pose Recognition in Parts from Single Depth Images," June 2011.

[28] D. Comaniciu, P. Meer, and S. Member, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, pp. 603–619, 2002.

[29] K. G. Derpanis, "Mean shift clustering." `http://www.cse.yorku.ca/~kosta/CompVis_Notes/mean_shift.pdf`, August 2005.

[30] H. Haas and A. Brown, "Web services glossary." `http://www.w3.org/TR/wsdl20-primer/`, June 2007.

[31] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer, "05462 service-oriented computing: A research roadmap," in *Service Oriented Computing (SOC)*, no. 05462, (Dagstuhl, Germany), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[32] OASIS, "Oasis soa reference model tc." `https://www.oasis-open.org/`, 2012.

[33] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz, "Reference model for service oriented architecture 1.0." `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`, October 2006.

[34] H. Voormann, "Web service." `en.wikipedia.org/wiki/Web_service`.

[35] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (wsdl) 1.0." `http://xml.coverpages.org/wsdl20000929.html`, September 2000.

[36] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (wsdl) 1.1." `http://www.w3.org/TR/wsdl`, March 2001.

[37] D. Booth and C. K. Liu, "Web services description language (wsdl) version 2.0." `http://www.w3.org/TR/wsdl20-primer/`, June 2007.

[38] D. Box, G. Kakivaya, A. Layman, S. Thatte, and D. Winer, "SOAP: Simple object access protocol." `http://tools.ietf.org/html/draft-box-http-soap-01`, November 1999.

[39] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap) 1.1." `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`, May 2000.

[40] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "Soap version 1.2." http://www.w3.org/TR/2003/PR-soap12-part1-20030507/, May 2003.

[41] B. Spies, "Web service to web service communication." http://ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest, May 2008.

[42] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[43] L. Richardson, "Justice will take us millions of intricate moves." http://www.crummy.com/writing/speaking/2008-QCon/, November 2008.

[44] M. Fowler, "Richardson maturity model: steps toward the glory of rest." http://martinfowler.com/articles/richardsonMaturityModel.html, March 2010.

[45] E. J. Gregorio, "The atom publishing protocol." http://www.ietf.org/rfc/rfc5023.txt, October 2007.

[46] O. Corporation, "The sun cloud api." http://kenai.com/projects/suncloudapis/pages/Home, September 2009.

[47] T. A. S. Foundation, "Apache couchdb." http://couchdb.apache.org/, July 2012.

[48] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big"web services: making the right architectural decision," in *17th international Conference on World Wide Web*, 2008.

[49] A. S. Foundation, "Apache mpm common directives." http://httpd.apache.org/docs/2.2/mod/mpm_common.html#maxclients, August 2012.

[50] I. Fette and A. Melnikov, "The websocket protocol." http://tools.ietf.org/html/rfc6455, December 2011.

Appendix A

Implementation

This chapter provides technical details about the prototype that I developed for evaluation purpose.

A.1 Data Accessing Layer Implementation

The data accessing layer is implemented in C++ and uses Microsoft NUI SDK to extract the data from the Kinect sensor. In this section, I will first start with the build-in data structures of color, depth and skeleton frame. Then I will show how to use NUI SDK functions to get access to those frames.

A.1.1 Data Frame Format

The image and depth frame share a mutual structure defined as:

```
typedef struct _NUI_IMAGE_FRAME
{
    LARGE_INTEGER             liTimeStamp;
    DWORD                     dwFrameNumber;
    NUI_IMAGE_TYPE            eImageType;
    NUI_IMAGE_RESOLUTION      eResolution;
    NuiImageBuffer *          pFrameTexture;
    DWORD                     dwFrameFlags_NotUsed;
    NUI_IMAGE_VIEW_AREA       ViewArea_NotUsed;
} NUI_IMAGE_FRAME;
```

- The **liTimeStamp** member indicates the elapsed time of the frame once the stream starts.
- The **dwFrameNumber** member counts the sequential number of the frame since the stream starts.
- The **eImageType** member defines which type of image the frame contains.

  If **eImageType** is equal to NUI_IMAGE_COLOR, the pointer **pFrameTexture** points to a normal 640 x 480 or 1280 x 960 three channel image captured by the standard color camera, where each pixel has three one-byte channels representing red, green, blue respectively(See Figure A.1).

  If **eImageType** is equal to NUI_IMAGE_DEPTH, then **pFrameTexture** refer to a 320 x 240 or 640 x 480 image captured by the range camera, where each pixel has two bytes with the lowest 11 bits storing the depth value(See Figure A.2).

Different from the image and depth frame, which are directly captured by the RGB camera and the range camera respectively, the skeleton frame is computed from each depth frame by the SDK runtime process using the method described in Chapter 3 (See Figure A.3).

The skeleton frame is defined in a NUI_SKELETON_FRAME structure as following:

```
typedef struct _NUI_SKELETON_FRAME
{
    LARGE_INTEGER          liTimeStamp;
    DWORD                  dwFrameNumber;
    DWORD                  dwFlags;
    Vector4                vFloorClipPlane;
    Vector4                vNormalToGravity;
    NUI_SKELETON_DATA      SkeletonData[NUI_SKELETON_COUNT];
} NUI_SKELETON_FRAME;
```

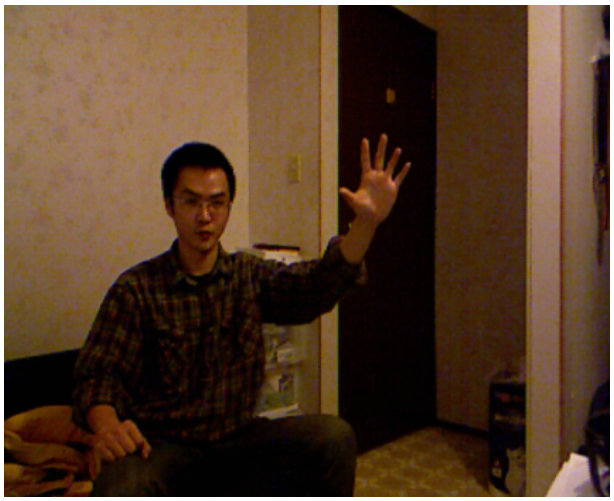


Figure A.1: RGB Frame

Figure A.2: Depth Frame

The **dwFrameNumber** member of the NUI_SKELETON_FRAME structure records the frame number of the depth frame from which the skeleton frame is computed. The SDK currently supports active tracking up to two skeletons simultaneously and passive tracking up to six.

The skeleton frame returns the data of all skeletons—whether it is tracked or not—in the **SkeletonData** member, which is an array containing six members of NUI_SKELETON_DATA structures defined in the following format:

Figure A.3: Skeleton Frame

```
typedef struct _NUI_SKELETON_DATA
{
    NUI_SKELETON_TRACKING_STATE eTrackingState;
    DWORD                       dwTrackingID;
    DWORD                       dwEnrollmentIndex;
    DWORD                       dwUserIndex;
    Vector4                     Position;
    Vector4             SkeletonPositions[NUI_SKELETON_POSITION_COUNT];
    NUI_SKELETON_POSITION_TRACKING_STATE
           eSkeletonPositionTrackingState[NUI_SKELETON_POSITION_COUNT];
    DWORD                       dwQualityFlags;
} NUI_SKELETON_DATA;
```

Each NUI_SKELETON_DATA is correspondent with a skeleton. The **eTrackingState** member indicates how the skeleton is tracked, which is in one of three possible values:

- If the skeleton is actively tracked, the **eTrackingState** is NUI_SKELETON_TRACKED, the **Position** vector indicates the mass center point of the skeleton, and the **SkeletonPositions** array is a set of 20 points in vector4 structure:

  ```
  struct{
    FLOAT x;
    FLOAT y;
    FLOAT z;
    FLOAT w;
  }
  ```

  Each (x, y, z) triple is correspondent with a critical joint's 3D position in the skeleton (see Figure A.4). And the index of skeleton points array is defined in an enumerator:

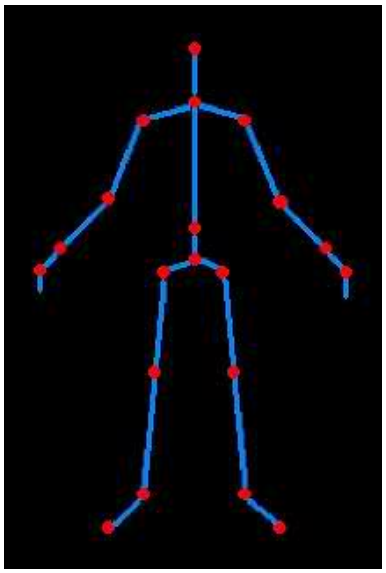


Figure A.4: Kinect Skeleton

```
typedef enum _NUI_SKELETON_POSITION_INDEX
{
    NUI_SKELETON_POSITION_HIP_CENTER = 0,  //Hip Center
    NUI_SKELETON_POSITION_SPINE,           //Spine Center
    NUI_SKELETON_POSITION_SHOULDER_CENTER, //Shoulder Center
    NUI_SKELETON_POSITION_HEAD,            //Head
    NUI_SKELETON_POSITION_SHOULDER_LEFT,   //Left Shoulder
    NUI_SKELETON_POSITION_ELBOW_LEFT,      //Left Elbow
    NUI_SKELETON_POSITION_WRIST_LEFT,      //Left Wrist
    NUI_SKELETON_POSITION_HAND_LEFT,       //Left Hand
    NUI_SKELETON_POSITION_SHOULDER_RIGHT,  //Right Shoulder
    NUI_SKELETON_POSITION_ELBOW_RIGHT,     //Right Elbow
    NUI_SKELETON_POSITION_WRIST_RIGHT,     //Right Wrist
    NUI_SKELETON_POSITION_HAND_RIGHT,      //Right Hand
    NUI_SKELETON_POSITION_HIP_LEFT,        //Left End of Hip
    NUI_SKELETON_POSITION_KNEE_LEFT,       //Left Knee
    NUI_SKELETON_POSITION_ANKLE_LEFT,      //Left Ankle
    NUI_SKELETON_POSITION_FOOT_LEFT,       //Left Foot
    NUI_SKELETON_POSITION_HIP_RIGHT,       //Right Hip
    NUI_SKELETON_POSITION_KNEE_RIGHT,      //Right Knee
    NUI_SKELETON_POSITION_ANKLE_RIGHT,     //Right Ankle
    NUI_SKELETON_POSITION_FOOT_RIGHT,      //Right Foot
    NUI_SKELETON_POSITION_COUNT
} NUI_SKELETON_POSITION_INDEX;
```

The **eSkeletonPositionTrackingState** array indicates whether the corresponding joint is actually being tracked or inferred from other joint poses.

- If the skeleton is passively tracked, the **eTrackingState** is equal to NUI_SKELETON_POSITION_ONLY. In this case only the center mass of skeleton is tracked, thus only the member **Position** contains valid value, while **SkeletonPositions** and **eSkeletonPositionTrackingState** do not.

- If the number of tracked skeletons is less than NUI_SKELETON_COUNT, which is equal to the max tracked skeleton number: 6, the **eTrackingState** member of those skeleton that not being tracked is

set to NUI_SKELETON_NOT_TRACKED and the values of all other members in the structure are invalid.

A.1.2 Data Frame Accessing

The SDK provides a set of functions which makes it possible for applications to access and manipulate data frames in an event-driven way. In general, data accessing process is a three-step procedure described below:

- Initiate the Kinect sensor by calling function **NuiInitialize**:

```
hr = NuiInitialize(
            NUI_INITIALIZE_FLAG_USES_DEPTH |
            NUI_INITIALIZE_FLAG_USES_SKELETON |
            NUI_INITIALIZE_FLAG_USES_COLOR);
```

  The above call tells the Kinect SDK we want to get image, depth and skeleton data and it will starts an internal thread to retrieve required data from Kinect sensor.

- Start streams and register each stream with a new frame notification event. Firstly create three event objects:

```
m_hNextDepthFrameEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
m_hNextVideoFrameEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
m_hNextSkeletonEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
```

  Then start three streams for color, depth and skeleton data respectively, and register the corresponding events with each stream:

```
//Start color stream
hr = NuiImageStreamOpen(
    NUI_IMAGE_TYPE_COLOR,
    NUI_IMAGE_RESOLUTION_640x480,
    0,
    2,
    m_hNextVideoFrameEvent,
    &m_pVideoStreamHandle );
//Start depth stream
hr = NuiImageStreamOpen(
    NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX,
    NUI_IMAGE_RESOLUTION_320x240,
    0,
    2,‘
    m_hNextDepthFrameEvent,
    &m_pDepthStreamHandle );
//Start skeleton stream
hr = NuiSkeletonTrackingEnable(
    m_hNextSkeletonEvent,
    0 );
```

  In this way, whenever a new frame in a stream is ready, the corresponding event will be set.

- Listening to the new frame notification and processing it.

  For each stream, create and start a thread listening to the corresponding new frame notification event:

```
//Threads name declaration
static DWORD WINAPI stream_depth_thread(LPVOID pParam);
static DWORD WINAPI stream_color_thread(LPVOID pParam);
static DWORD WINAPI stream_skeleton_thread(LPVOID pParam);
//Start threads
_pColorThreadHandle    = CreateThread(NULL, 0,
                                      Nui_draw_color_thread,
                                      this, 0,  NULL);
_pDepthThreadHandle    = CreateThread(NULL, 0,
                                      Nui_draw_depth_thread,
                                      this, 0, NULL);
_pSkeletonThreadHandle = CreateThread(NULL, 0,
                                      Nui_draw_skeleton_thread,
                                      this, 0, NULL);
```

within each thread, there is a sequence like:

```
while(true){
        WaitForEvent(m_hNextDataFrameEvent);
        DealWithNextDataFrame();
}
```

In this way, whenever a new data frame comes, the related listening thread will be awaken from the pending state and process each type of data frame using one of the data-processing functions:

```
void dealWithNextImageFrame(KinectObserver* dev);
void dealWithNextDepthFrame(KinectObserver* dev);
void dealWithNextSkeletonFrame(KinectObserver* dev);
```

A.2 Gesture Recognition Layer Implementation

As specified in the architecture definition, the gesture recognition layer is implemented as an observer to both depth and skeleton frame updates. When the recognition process is conducted in real-time along with frame generation, in order to avoid any delay, the whole process need to be finished within the intervals between two successive frame updates e.g. 33ms. To meet with such requirement, I have designed a fast hand gesture recognition pipeline involving two steps: First a hand model is generated from depth frame using a set of fast image processing algorithms, then the hand model will be input into a simple yet robust recognition pipeline based on automaton to generate the gesture events. Details about my method is described as follows.

A.2.1 Hand Model Generation

The idea behind our hand model generation method can be simply described as:

- Using the hand position in the skeleton frame to locate the hand in the depth frame.
- Analysing the hand region in the depth frame to extract hand status information

Our hand model generation process is a pipeline of three steps consisting of **hand segmentation**, **figure tip detection** and **figure tip classification**.

Hand Region Segmentation

Given a hand joint's 3D coordinate (X, Y, Z) (in meters) in the skeleton frame, we could get a 40cm x 40cm virtual rectangle defined by four vertices: $(X-0.2, Y-0.2, Z), (X+0.2, Y-0.2, Z), (X+0.2, y+0.2, Z), (X-$

$0.2, y+0.2, Z)$, which is parallel to the camera and guarantees to enclose the hand. By triangulation formula:

$$X' = 0.5 + \frac{X \times F}{Z \times R_x}, \tag{A.1}$$

$$Y' = 0.5 + \frac{Y \times F}{Z \times R_y}, \tag{A.2}$$

where R_x and R_y is the depth frame's resolution along X, Y axis, f is the range camera focal length measured in pixel, we can compute the diagonal $[(X-0.2, Y-0.2, Z), (X+0.2, Y+0.2, Z)]$'s projection on the depth frame $[(X'_1, Y'_1), (X'_2, Y'_2)]$, which determines a depth invariable window around the hand in the depth frame (See Figure A.5c).

(a) Skeleton **(b)** Depth Image **(c)** Hand Region

Figure A.5: Locate Hand Position in Depth Image

Then in the hand window, by applying a binary range threshold: $(Z-0.4, Z+0.1)$, we could filter any pixel whose depth value is more than $40cm$ behind or $10cm$ in front of the hand joint point to segment out the hand region (See Figure A.6b). On the basis of a binary hand segmentation, it is straightforward to extract the hand contour by calling **OpenCV** function *cvFindContours()*, which will sort and store all pixels on the contour to an array in clock-wise direction (See Figure A.6c).

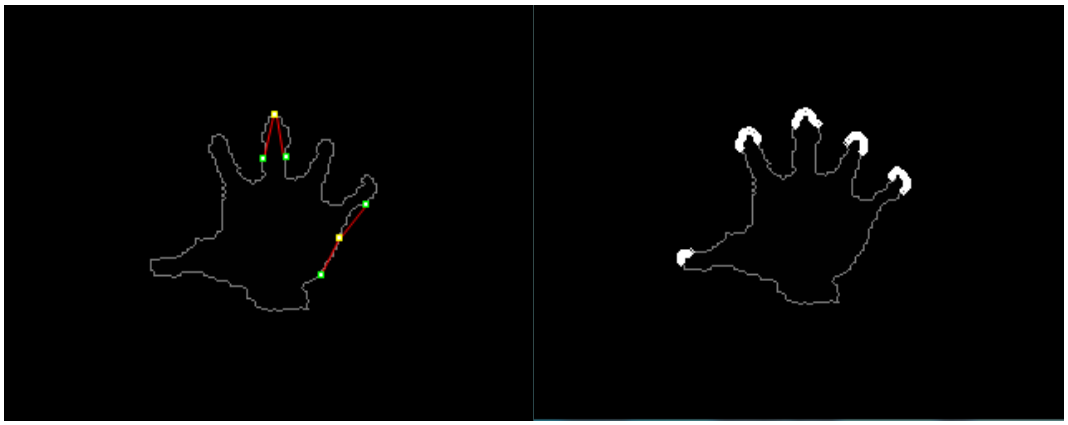
(a) Hand Region **(b)** Thresholded Hand Region **(c)** Hand Contour

Figure A.6: Hand Contour Extraction

Finger Tip Detection

Once we obtain the hand contour, we can start doing analysis on it to determine the finger tip positions. What we do is as follows: We first apply a curvature filter to the contour, which is called K-Curvature method: For each pixel p_i on the hand contour, in clock-wise direction, we pick up the kth pixel before it: p_{i-k} and the the kth pixel after it: p_{i+k}. If the angle $\angle p_{i-k} p_i p_{i+k}$ is below a certain threshold θ, which indicates

something dramatically bumps out, we consider the center pixel as a finger tip candidates, otherwise not (See Figure A.7).

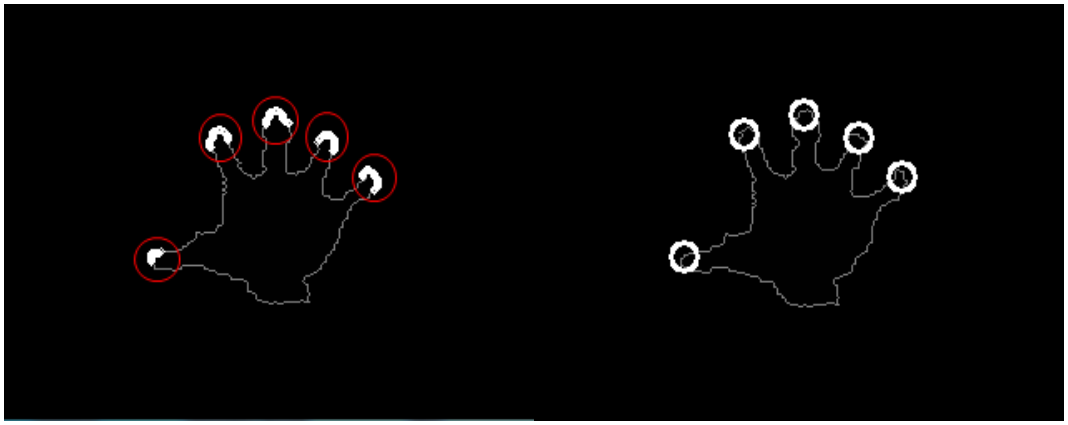


(a) K-Curvature Filtering **(b)** Finger Tip Candidates (High Lighted)

Figure A.7: K-Curvature Filtering Process

After the initial curvature filtering, on each finger tip, there will be multiple finger tip candidate pixels (See Figure A.7b). Then I use a simple aggregation algorithm to determine the position of each finger tip:

- First of all, iterate the contour from the bottom-most pixel in clock-wise direction. For each candidate pixel, if it is less than n pixels away from the previous candidate pixel, classify it into the same group as the previous pixel. Otherwise, create a new group and pack the pixel into it (See Figure A.8a).
- For each group obtained from the previous step, if the pixel number in it exceeds a certain threshold t, the group will be considered as a figure tip and its position will be computed by averaging all pixels' coordinates in it (See Figure A.8b).



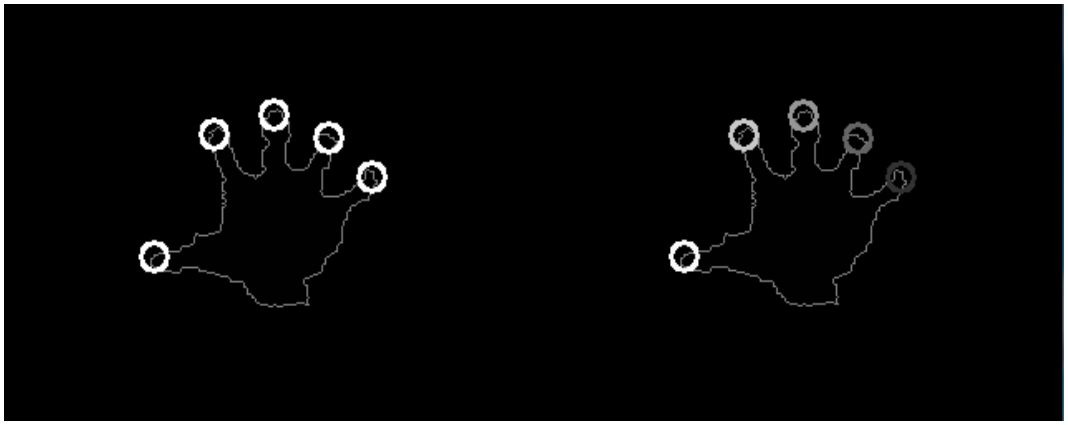
(a) Finger Candidates Group **(b)** Finger

Figure A.8: Finger Tip Detection Process

Finger Tip Classification

When the finger tip positions are determined, the next step is to discriminate different fingers. To avoid creating heavy computational burden, we use a light-weighted tracking based approach to tackle the problem:

- First of all, the user needs to register his/her hand with a full open hand, where all five finger tips need to be detected (See Figure A.9a). In this case, the discriminating task is easy: Take the left hand as example, we simply assign each finger tip with a index number from 0 to 4 in clockwise direction(See Figure A.9b).



(a) Full Hand **(b)** Indexed Full Hand

Figure A.9: Finger Tip Detection Process

- In the subsequent frames, if there is one or more finger tips missing, then each finger tip shown will be automatically mapped to the nearest one in the previous frame(See Figure A.10). In this way, we can quickly and robustly distinguish each finger tip.

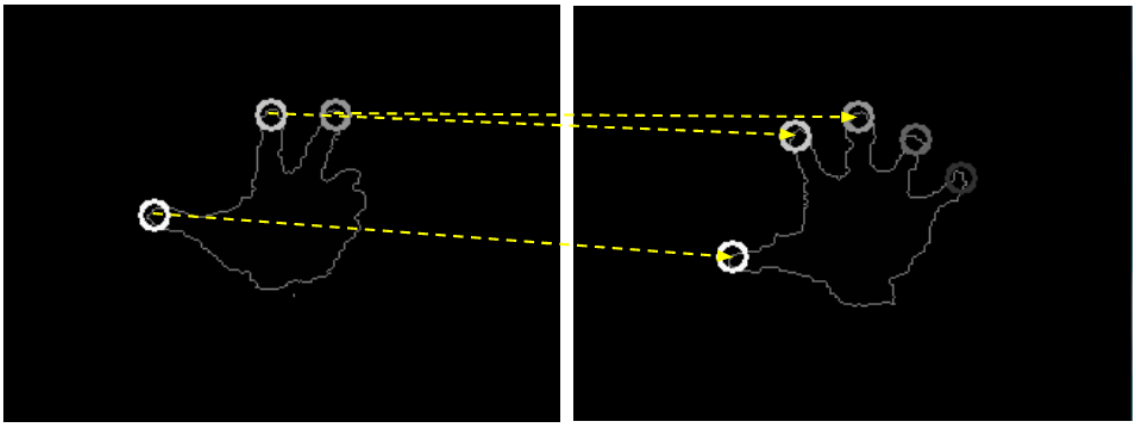


Figure A.10: Nearest Mapping based Finger Tip Classification

A.2.2 Gesture Representation & Recognition

With all the five finger tips detected and distinguished, the status of a hand can be simply represented by a frame structure as:

```
typedef struct HandStatusFrame{
    bool figureTips[5];
}
```

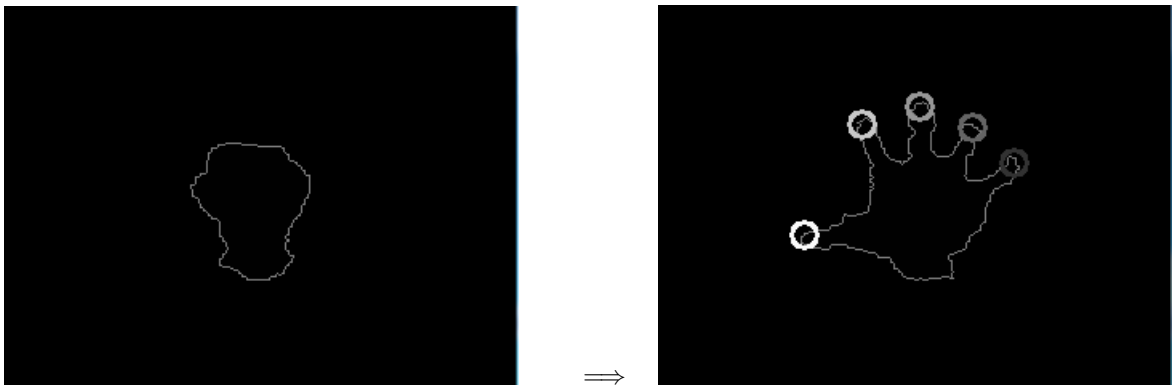
Where ***fingerTips[]*** is a five-element boolean array correspondent with five finger tips in clockwise direction, with each element's value (true or false) indicating whether the corresponding finger tip is extending out or not. In this way, a hand gesture can be defined as a sequence of key frames aligned in a chronicle order as:

```
#define HandGestureSequence vector<GestureFrame>
```

where each key frame ***GestureFrame*** is defined as:

```
typedef struct GestureFrame{
    HandStatusFrame status;
    int timeRange[2];
}
```

where ***status*** is a ***HandStatusFrame*** and ***timeRange*** is a two-element interger array defined a time interval (in milliseconds). This two-element tuple means the hand status specified by ***status*** is expected to show up during the interval specified by ***timeRange*** after the previous frame in the gesture sequence. For example, a simple open hand gesture can be defined by a two-frame gesture sequence as Table A.1:



fingerTips[5]:$\{false, false, false, false, false\}$
timeRange[2]: $[0, \infty]$
Frame1

$\Longrightarrow$

fingerTips[5]:$\{true, true, true, true, true\}$
timeRange[2]: $[0, 100]$
Frame2

Table A.1: Open Hand Gesture Defined by two Key Frames

When a gesture sequence is defined, the gesture recognition process can be naturally accomplished by a **Deterministic Finite Automaton (DFA)**: For each gesture defined by a sequence with n key frames, the system can automatically generate a $n+1$ states DFA $M = (Q, \sum, \delta, q_0, F)$ (See Figure A.11), consisting of

- a states set: $Q = \{s_0, s_1, ..., s_n\}$,
- a possible inputs set: $\sum = \{a_1, a_2, ..., a_n, \hat{a}_1, \hat{a}_2, ..., \hat{a}_n\}$, where a_i $(i \in \{1, 2, ..., n\})$ means the action specified in frame i happens while $\hat{a}_i$ means any action other than a_i and a_1,
- a state transition function: $\delta : Q \times \sum \rightarrow Q$ defined as:

$$\delta(s_i, a) = \begin{cases} s_{i+1} & \text{if } a = a_{i+1} \\ s_1 & \text{if } a = a_1 \\ s_0 & \text{if } a \in \hat{a}_{i+1} \end{cases} \tag{A.3}$$

- a start state: $s_0 \in Q$,
- a set of accept states: $F = \{s_n\}$.

To put it simply, assuming we are now on each state s_i, if the system detects the hand status specified by the frame f_{i+1} showing up during the specific time range, the automaton will enter the next state, otherwise

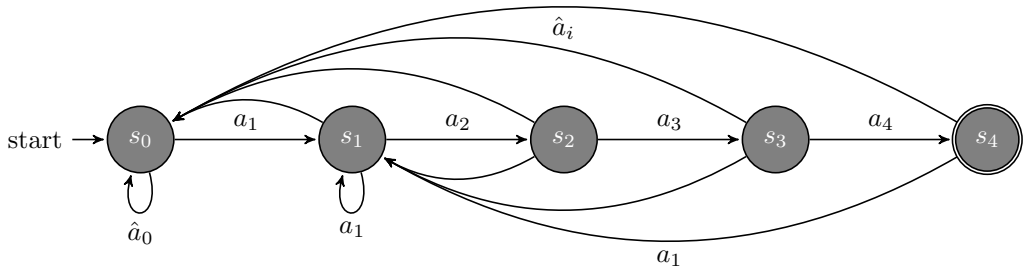


Figure A.11: Automaton for a Gesture defined by 4 Key Frames

it will return to s_0 or s_1 depending on whether the input is $\hat{a}_i$ or a_1 and start over again. When the automaton reaches the accepting state s_n, the corresponding gesture will be recognized as performed. Once the automaton is generated, we can represent gesture in an automaton form as:

```
typedef struct GestureAutomaton{
    int statesNum;
    int currentStateIndex;
    HandGestureSequence keyFrames;
}
```

Where ***statesNum*** specifies the total number of the states in the automaton, ***currentStateIndex*** indicates which state it is now in and ***keyFrames*** defines the input expected by each state to enter the next one.

A.3 Interface Layer

The interface layer is formed by four similar classes named ***KinectColorSender***, ***KinectDepthSender***, ***KinectSkeletonSender***, ***KinectGestureSender***, which are responsible to accept TCP connection requests from HTTP Server Layer and sending out the related data through those connections. To be able to accept new TCP connections, in every class's initiation method, a connection accepting thread like

```
DWORD KinectImageSocket::socket_accepting_thread(LPVOID pParam){
        KinectImageSocket *handle = (KinectImageSocket *) pParam;

        sockaddr_in from;
        int fromlen = sizeof(from);
        while(handle->_acceptingSign){
                SOCKET *client = new  SOCKET();
                *client = accept(handle->_s, (sockaddr*)&from, &fromlen);

                if(*client != INVALID_SOCKET){
                        handle->_clientSockets.push_back(client);
                        handle->_isThereClient = true;
                        printf("new connection updated\n");
                }else
                        delete client;
        }
        return true;
}
```

will be started, which will listen to the incoming TCP connection requests on certain predefined port, whenever a request comes, a new connection socket will be created and pushed into a socket queue. Meanwhile, since as each of them is implemented as an observer listening to the related updates, when the updates/events are triggered, it will be notified and send out the updated frame/events to every element of the sockets queue in its ***update()*** method like:

```
// For each element in socket queue
for(int i = 0; i < _clientSockets.size(); i++){
     int bytesSent = 0;
     int bytesYetTobeSent = _frameLengthInByte;
     printf("Sending Image data\n");
     // Make sure the entire frame is sent out
     while(bytesYetTobeSent > 0 && _isThereClient) {
            int temp= send(*_clientSockets[i],
                           (char*)(_imageFrame + bytesSent),
                           bytesYetTobeSent, 0);
         bytesSent += temp;
         bytesYetTobeSent -= temp;
        }
        printf("Image data is sent\n");
}
```

A.4 HTTP Server Layer

In my prototype, I use Apache Tomcat to host HTML pages and implemented a Java module to communicate with the Motion Sensor Server via TCP connections. In the Java module there are four receiver classes named ***ColorFrameReceiver***, ***DepthFrameReceiver***, ***SkeletonFrameReceiver*** and ***GestureEventReceiver***. They are connected to receive data from four sender classes in the Interface Layer respectively. Then these data received by different receivers will be stored as different local objects ***colorFrame***, ***depthFrame***, ***skeletonFrame*** or ***gestureEvents*** and are ready to be send to the clients. My prototype implements both pushing and polling data access models described in the architecture chapter:

- For the pushing model, every receiver class will open up a Websocket waiting for connections from the clients. And in every data resource's stream HTML page, there is a piece of Javascript code like:

  ```
  var ws = new WebSocket("ws://MOTION_SENSOR_SERVER_IP:PORTNO/");
  ```

  to initiate the Websocket connection. In this way, whenever the client's web browser receives this page and executes the script, a Websocket connection will be established to the corresponding socket on the server and the data will be pushed to the client subsequently.

- For the polling model, I create four servlet classes ***GetColorFrameServlet***, ***GetDepthFrameServlet***, ***GetSkeletonFrameServlet***, ***GetGestureEventServlet*** as bridges between corresponding HTML pages and receiver classes. Whenever the client receives a snapshot web page which contains an URL to a servlet class, the client browser will send a HTTP request again to the servlet URL, which will trigger the servlet activity to fetch and return a data frame that will finally be embedded and visualized in the HTML page.