G. C. Fox, S. Kamburugamuve and R. D. Hartman, "Architecture and measured characteristics of a cloud based internet of things," *2012 International Conference on Collaboration Technologies and Systems (CTS)*, Denver, CO, 2012, pp. 6-12.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6261020&isnumber=6261004

Abstract:
The Internet of Things (IoT) many be thought of as the availability of physical objects, or devices, on the Internet [1]. Given such an arrangement it is possible to access sensor data and control actuators remotely. Furthermore such data may be combined with data from other sources - e.g. with data that is contained in the Web - or operated on by cloud based services to create applications far richer than can be provided by isolated embedded systems [2,3]. This is the vision of the Internet of Things. We present a cloud-compatible open source controller and an extensible API, hereafter referred to as `IoTCloud', which enables developers to create scalable high performance IoT and sensor-centric applications. The IoTCloud software is written in Java and built on popular open source packages such as Apache Active MQ [4] and JBoss Netty [5]. We present an overview of the IoT Cloud architecture and describe its developer API. Next we introduce the FutureGrid - a geographically distributed and heterogeneous cloud test-bed [6,7] - used in our experiments. Our preliminary results indicate that a distributed cloud infrastructure like the FutureGrid coupled with our flexible IoTCloud framework is an environment suitable for the study and development of IoT and sensor-centric applications. We also report on our initial study of certain measured characteristics of an IoTCloud application running on the FutureGrid. We conclude by

inviting interested parties to use the IoTCloud to create their own IoT applications or contribute to its further development.

Since the standardization of the TCP/*IP* protocol suite over thirty years ago, the Internet has grown steadily from a collection of small research networks to a ubiquitous worldwide network of interconnected networks that serviced nearly two billion unique users in 2009 [8]. Now with the widespread availability of wireless Internet connectivity combined with the low cost of miniature electronic devices it possible to imagine the Internet expanded to include objects, embedded with sensors, communicating over the Internet in in vast numbers [9]. These smart objects are ordinary physical things which are augmented by a small computer that includes a sensor or actuator and a communication device [3]. A smart object is thus an embedded system, consisting of a thing (the physical entity) and a component (the computer) that processes the sensor data and supports a, usually, wireless communication link to the Internet [2].

Smart objects can be battery-operated, but not always, and typically have three components: a CPU (8-, 16- or 32-bit micro-controller), memory (a few tens of kilobytes) and a low-power wireless communication device (from a few kilobits/s to a few hundreds of kilobits/s). The size is small and the price is low: a few square mm and few dollars [10].

The technical issues involved with the loT vision are not related to the functional capabilities of smart objects - there many such embedded systems are connected to the Internet today - but in the potentially massive numbers of smart objects to be contended with. Examples of these issues include: the deployment, discovery, management and interoperability of a many types (varieties) of smart objects deployed in large quantities. Therefore, there is a need for scalable systems which enable the exploration of these and the other related technical issues associated with creating loT applications. To this end we have developed a sensor-centric framework called the loT Cloud which supports an extensible set of sensor-types and large numbers of, possibly, geographically distributed smart objects. The primary purpose of this paper is to introduce the IoTCloud and encourage you to try it. We will also discuss the suitability of distributed clouds for hosting our middleware and present an initial measurement of system

performance in the context of a simulated real-time video chat application.
The rest of this paper is organized as follows. Firstly, we present the architecture of our IoTCloud framework and describe the API available to end-users. Secondly, we describe some of the sensors and applications we have implemented using with our framework. Thirdly, we give an overview of the underlying heterogeneous, distributed, cloud infrastructure called the FutureGrid [6][7] used as a test-bed during IoTCloud development. Next, we discuss an experiment measuring IoTCloud performance in several scenarios. Finally, we present our conclusions and ideas for future work.

SECTION II.

IoTCloud Architecture

We have adopted an open architecture for the loT Cloud framework which, from a bird's-eye view, consists of four primary components:

1 IoTCloud Controller
2 Message Broker
3 Sensors
4 Clients

The *loTCloud Controller* is responsible for managing the other system components and providing SOAP Web Services for sensor registration, discovery, subscription and control. The *Message Broker* handles the low level details of message routing. The combination of the *Controller* and the Message Broker provides a middleware layer for the system. *Sensors* may be smart objects or computational services. In fact anything producing a time dependent data series may be considered as a sensor. *Clients* subscribe to (consume) sensor data for some application specific purpose. It is, of course, possible for an object to be both a *Sensor* and *Client* in this model. Figure 1. shows relationship between the basic IoTCloud components.

Workaround for combined images.Eg.- 1000116 Fig. 5

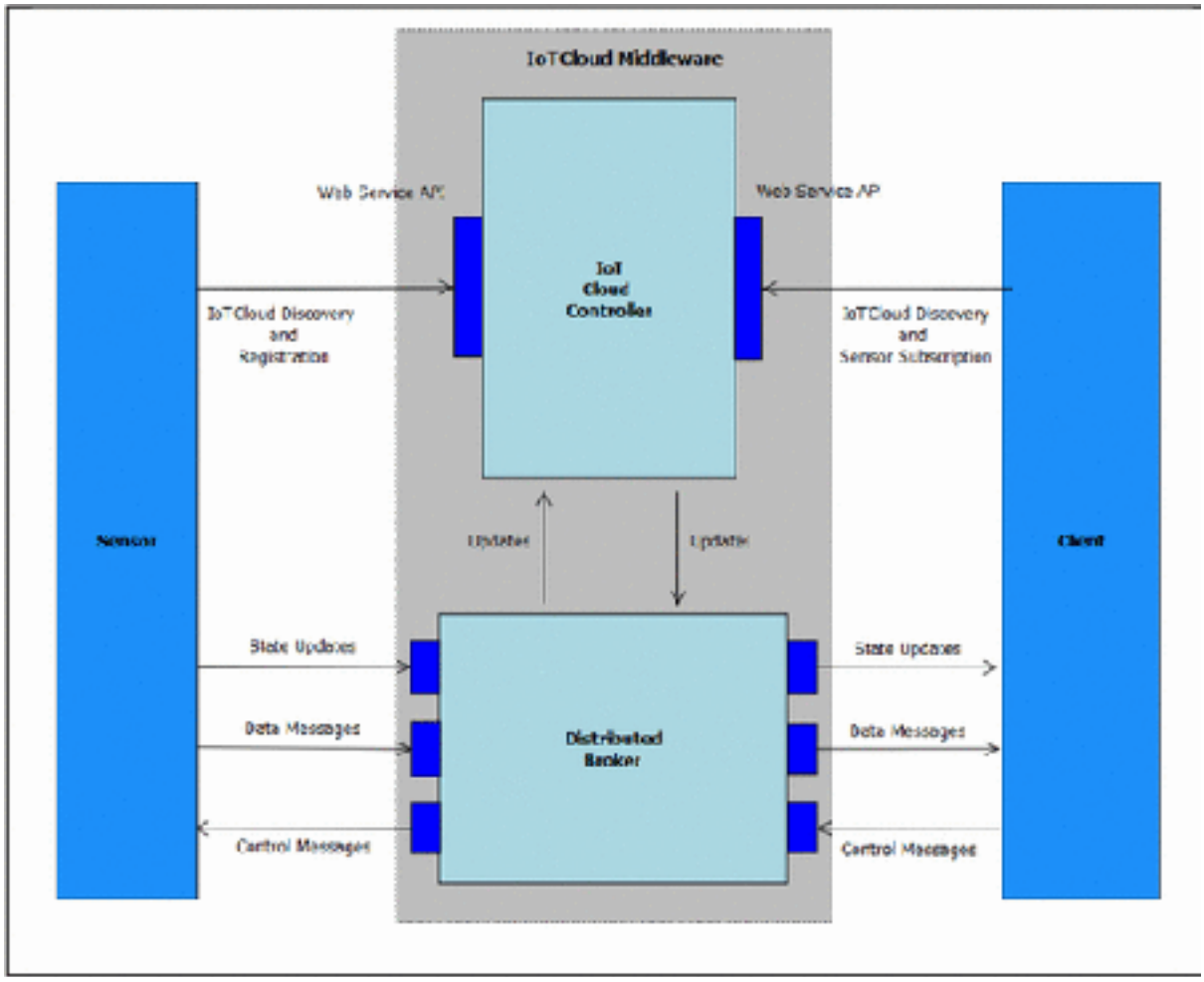


Figure 1. IoTCloud Architecture

View All

A. IoTCIoud Controller

The primary functionality of *1oTCIoud Controller* is to coordinate communication between the other components and provide system management services. The *Controller* uses the Message Broker to create message routes (e.g. JMS topics) between clients and sensors. It also maintains a repository of sensor status and metadata information used for discovery, filtering, and management services. For example, if a client wishes to know the names of any sensors online at particular location that information is available from the controller. In addition to sensor status and metadata information the *Controller* provides services allowing: sensors to publish data, clients to subscribe to it and, possibly, send control messages back. These services are carefully constructed to shield publishers and subscribers from the details of the underlying message transfer mechanisms.

IoT applications inherently involve connecting various types of devices together and as such interoperability is a key design goal for us. Interoperability is achieved by having sensors and clients interact with the IoTCloud through a set of standards-based SOAP Web Services [11]. These web services are written using Apache Axis2, hosted in a JBoss Jetty server, and designed to make the process of implementing new sensors and clients a straight forward one.

B. Message Broker

In general sense, we use the term “message broker” to refer a system that accepts messages and distributes them to set of subscribers. We use a JMS [12] style message broker to handle block type data and a streaming message broker to handle streaming data. All the routing rules and configuration of the broker are managed by the Controller and the specific details of the broker are hidden from the client and sensor developers.

C. JMS Message Broker

This broker is used for distributing block sensor data as well as all of internal the update and control messages. The loTCloud uses Apache Active MQ [4] for its *JMS Message Broker*. Active MQ is compatible with the JMS 1.1 specification and one of the most popular open source messaging servers currently available. The *JMS MessageBroker* is typically used were asynchronous message delivery is appropriate, e.g. GPS data.

D. Streaming Message Broker

Our *Streaming Message Broker* is a simple low footprint HTTP server written using the Netty [5] technology. Netty is an appropriate choice since its memory usage is constant, i.e. determined by message size and not the number of client connections, and based on Java the NIO API for intensive I/O operations. The *Streaming Message Broker* is well suited for real-time sensor data, e.g. video streaming.

E. Sensor Module

A Sensor Module is a software component which publishes a time dependent series of data messages to the IoTCloud. A Sensor Module provides linkage between actual (physical) sensors and the IoTCloud. A typical example is a GPS Sensor Module. A user trying to connect a GPS device to the loTCloud would deploy a GPS Sensor Module that: a) registers itself with the IoTCloud and b) establishes

communication (over Bluetooth, USB, wireless, etc.) with the physical GPS device.
There are two major types of Sensor Modules: Block or Streaming. Block Sensors publish data via the JMS Message Broker which delivers it asynchronously. Block Sensors Modules are well suited for physical sensors with low (e.g. ~ 1 Hz.) publication frequencies, as well as physical sensors those individual data packets may be processed independently. GPS, RFID, thermometers, barometers etc. Data from Streaming Sensors is handled by the Streaming Message Broker and supports connection-oriented message delivery. Video is the canonical example of a Streaming Sensor but this sensor type is also useful whenever message delivery order is important.
Apart from sending data, Sensor Modules may also listen for control messages. Control messages are sent by the loT Controller on the behalf of Clients and carry sensor specific information about operations that the sensor should perform. An example of this is could be the pan, tilt and zoom functions of a Web Camera sensor.

F. Clients

The IoTCloud employs a loosely-coupled architecture based on message-oriented communication between *Clients* and *Sensors*. In such an arrangement applications may fire-and-forget messages to a broker that manages the details of message delivery. This is an especially powerful benefit in heterogeneous environments, allowing *Clients* and *Sensors* to be written using different languages and even possibly different wire protocols.
Clients are able to discover, subscribe to and control Sensors by using the Web Service API provided by the IoTCloud controller. Note an individual software component may be both a Client and a Sensor. A Video Face Detector subscribes to a Video Sensor, analyzes the video stream looking for human faces, and then publishes its own video stream which is the original video now augmented with circles drawn around any faces appearing on screen.

G. Update Distribution

When a client joins the IoTCloud it discovers set sensors currently available to it. Clients are able to register with the Controller to receive a filtered list of sensors of interest, e.g. a client may only wish to see video sensors at a certain location. Furthermore since the sensors and clients may join the IoTCloud in an ad-hoc manner

client also receive (through a callback function) status updates informing then when sensors of interest join or leave the IoTCloud. Our update distribution mechanism employees a ‘push’ model where the Controller automatically notifies clients of any new sensor events of interest.

H. Application Programming Interface (API)

From an end-users perspective, the important APls are the Sensor API and Client API. The Sensor API is designed to facilitate easy development of sensors modules and data publication. The Sensor API allows end-users to define a sensor-specific data format and a set of control messages for their custom sensor modules. The Client API is designed to facilitate the easy consumption of sensor data, sending of control messages and sensor discovery. Both these API's are currently available as Java jar class libraries. However, as we are using standards based technologies like SOAP Web Services, these API's can easily be extended to support other languages.

I. Sensor API

The Sensor API provides routines to register sensor modules with the IoTCloud, publish data, and receive control messages sent by either the middleware or clients.

First, a sensor must initialize the IoTCloud libraries by pointing the required bootstrap configuration files. These configuration files include the necessary networking information to connect to the IoTCloud. Next, it registers itself to the IoTCloud by providing its name, type of the sensor and group. Sensor ‘type’ denotes whether it is a block sensor or a streaming sensor.

Once registered, it can begin publishing data. As data is captured by a sensor module from a physical sensor (or other data source) it is wrapped with message envelope and submitted to the IoTCloud by functions provided in the Sensor API.

After starting its operation, sensors may receive control information from the middleware. It is up to the sensor implementer to act upon this control data.

Finally upon completing its operation a sensor should notify the middleware system it is terminating. The middleware system also performs periodic heath checks on all registered sensors. If a sensor fails to respond within a given time, the Controller assumes that the sensor is no longer available and closes the communication paths.

J. Client API

The *Client API* consists of several classes for registering a client with the IoTCloud and subscribing to sensors of interest. In the typical sequence client registration begins with initializing the loTCloud libraries by pointing to the correct configuration files. This is process the similar to that of a sensor module.
Next the client registers itself with the IoTCloud by pointing to the loTCloud URL. Here the underlying APls query the Controller for the networking information necessary to connect to the IoTCloud. For example the client will receive information about the transports that are supported by the controller. Here the ports of the HTTP transport or the JNDI properties of the JMS transport are sent through this call. This information is used internally by the client IoTCloud libraries and it not required in the client developer.
Once clients are registered with the Controller, they may request information about the sensors which are connected to the system. Various filters may be applied the request to narrow the scone of interest.
With names or the ID's of the sensors meeting the client's selection criteria, the client may subscribe any sensors of interest. The data is delivered to the user code by either the JMS or Streaming message brokers as appropriate. At this point clients may also issue sensor-specific control messages to the set of sensors it subscribes to.
Finally when the client is no longer interested in receiving data, it may unregister itself from the IoTCloud Controller.

K. Message API

IoTCloud has two broad categories of messages: *control messages* and *data messages*. The Client API and Sensor API have access these both message types.
A control messages is a message with a command id and a dataset of key value pairs specific to that command. The definition of normal control messages are user defined and implemented by the creator of a given sensor module. However, there are some system control messages that IoTCloud explicitly uses for controlling the behavior of sensors in certain predefined situations like when shutting down the IoTCloud. Control messages are typically sent from the clients (or the Controller) to sensors.
Similarly the formats of data messages are defined at design time and describe the specific data payload being published by a given sensor type. Data messages comprise the bulk of IoTCloud messaging and represent sensor data.

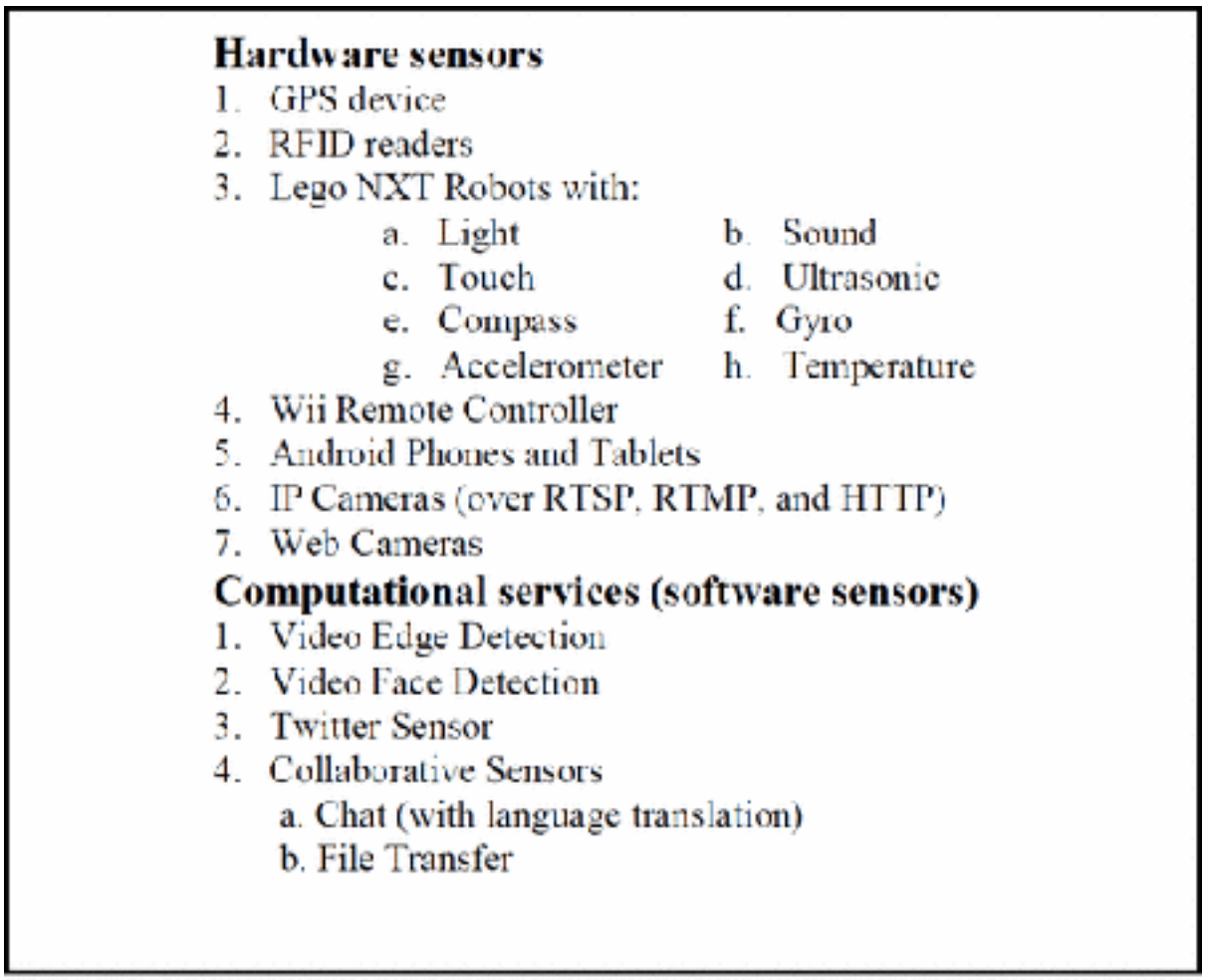
Since each sensor module potentially publishes a uniquely formatted data messages with different semantics the IoTCloud it does not attempt to infer or act upon the message itself. For the middleware a message is just a data block or a stream. Message schema is strictly a contract between the sensors and its consumers. It is up to the sensor and the client programmer to decide which data formats to subscribe to and how to handle those data messages. However since all loTCloud are messages derived from a common type it is possible to place messages from any sensor in a generic collection, or interact with properties common to all loTCloud messages such at timestamp. Details of the message type may also be determined at runtime through reflection.

SECTION III.

IoTCloud Sensors and Clients

The IoTCloud framework is an outgrowth of previous work in publish/subscribe based messaging systems done by the Community Grids Laboratory over the last ten years [13][14][15] [16] and with it we provide sample implementations for several interesting sensors and client applications.

Our system considers a sensor generically as a producer of time dependent data streams, making it flexible enough to support both hardware (physical) and computational (software) sensors. Some examples of currently supported sensors include:



Selected Set of IoTCloud Clients/Sensors

Future work is currently underway to build Sensor Modules for video cameras mounted on Quad or Hexa Copters [17] and Microsoft Kinect [18] sensors on TurtleBots [19]. See Figures 2. and 3.

Workaround for combined images.Eg.- 1000116 Fig. 5

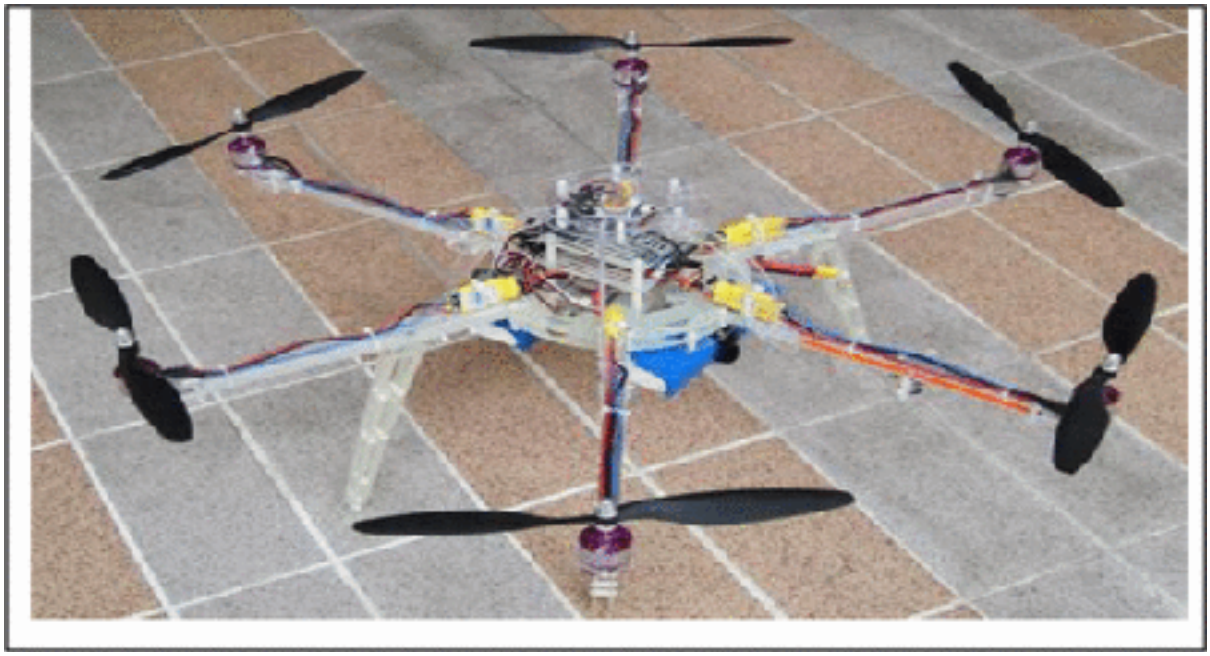


Figure 2. HexaCopter from jDrones

View All

Workaround for combined images.Eg.- 1000116 Fig. 5

Figure 3. TurtleBot from Willow Garage

View All

SECTION IV.
Futuredgrid

FutureGrid [7] is a part of the Extreme Science and Engineering Discovery Environment (XSEDE) [20]. The FutureGrid provides a capability that makes it possible for researchers to tackle complex research challenges in computer science related to the use and security of grids and clouds. These include topics ranging from authentication, authorization, scheduling, virtualization, middleware design, interface design and cybersecurity, to the optimization of grid-enabled and cloud-enabled computational schemes for researchers in astronomy, chemistry, biology, engineering, atmospheric science and epidemiology [6]. The project has several computing clusters at different locations with a sophisticated virtual machine and workflow-based simulation environment to support research on cloud computing, multicore computing, new algorithms and software paradigms.
Unlike production cloud systems like the Amazon EC2, Microsoft Azure or Google App Engine for commercial applications, or XSEDE [20] for scientific computing, FutureGrid, by contrast, is oriented towards developing tools and technologies rather than providing production computational capacity [21].
FutureGrid is an infrastructure comprising currently approximately 4,000 cores at six sites - Indiana University (11 Teraflop IBM 1024 cores, 7 Teraflop Cray 684 cores,
5 Teraflop Disk Rich 512 cores), University of Chicago (7 Teraflop IBM 672 cores), University of California San Diego Supercomputing Center (7 Teraflop IBM 672 cores), University of Florida (3 Teraflop IBM 256 cores),
Purdue University (4 Teraflop Dell 384 cores) and Texas Advanced Computing Center (8 Teraflop Dell 768 cores) - connected by a high-speed, network which is dedicated except for public link to Texas Advanced Computing Center. It is an experimental test-bed that could support large-scale research on distributed and parallel systems, algorithms, middleware and applications. Figure 4. shows the connectivity of the six sites.
Workaround for combined images.Eg.- 1000116 Fig. 5

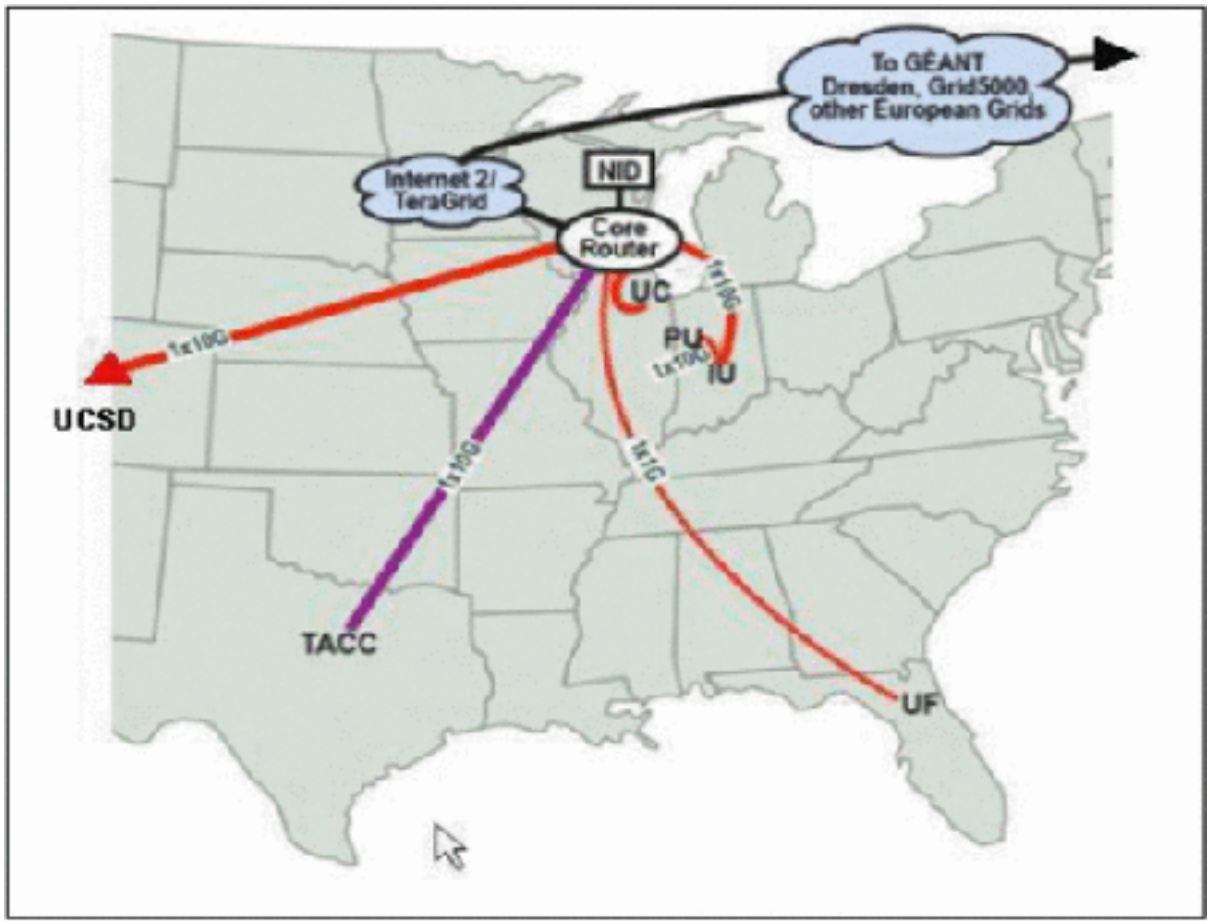


Figure 4. FutureGrid Connectivity

View All

FutureGrid includes services accessible to users to run HPC (High Performance Computing) jobs such as MPI, or Hadoop. It also supports several popular grid and cloud environments including the Eucalyptus, Nimbus and OpenStack clouds.

Eucalyptus [22][23] is an open source software platform that implements an Infrastructure-as-a-Service (IaaS)-style cloud computing. Eucalyptus provides an Amazon Web Services (AWS)-compliant, EC2-based web service interface for interacting with the cloud service. Additionally, Eucalyptus provides Walrus, an AWS storage-compliant service, and a user interface for managing users and images.

Nimbus is an open source toolkit that allows one to turn a cluster into an IaaS cloud [24]. Nimbus on FutureGrid allows users to run virtual machines on FutureGrid hardware. A Nimbus account user can easily upload custom-built virtual machine (VM) image or customize an image provided by FutureGrid. When a VM is booted, it is assigned a public IP address (and/or an optional private

address). The VM is accessible by logging in as root via SSH. A user can then run services, performs computations, and configure the system as desired. After using and configuring the VM, the modified VM image can be saved to the Nimbus image repository. OpenStack is an open source, IaaS cloud computing platform established by Rackspace Hosting and NASA and widely used in industry [26]. It includes three components: Compute (Nova), Object Storage (Swift) and Image Service (Glance). OpenStack is also fully Amazon EC2 complaint and supports an (AWS)-complaint web interface. OpenStack images are manipulated with the familiar eucazools [26]. Our IoTCloud experiments were performed using OpenStack for virtual machine deployment/management on FutureGrid resources.

SECTION V.

Performance Characteristics

In previous experiments we have investigated middleware performance at the network, message, and application level [27] [28][29]. We found that a pub/sub based middleware is an appropriate model for scalable sensor-centric, collaborative and loT applications. In this section we investigate the performance characteristics of a simulated real-time video chat application implemented using the IoTCloud framework and deployed with a single (i.e. non-distributed) message broker. We selected the popular TRENDnet TV-IP422WN IP camera as our baseline [30]. The TV-IP422WN camera publishes audio/video data over an RTSP stream at a rate of approximately 1800kbps when using the following encoding:

V.

Video

codec MPEG4: width: 640: height: 480: format: YUV420P; frame-rate: 30 frames/sec

V.

Audio

codec PCM MULAW; sample rate: 8000; channels: 1; format: FMT_S16

In order to simulate video sensors of this type we publish randomized data in 7680 byte packets at a rate of 30 packets per second. It is worth noting that this frame rate and packet size is also reasonable approximation of the Microsoft Kinect sensor [18] which we plan to conduct future experiments with.
Our software was deployed on the FutureGrid using a single OpenStack *mI. large* instance [31] to host the middleware and the simulated sensors. Subscribers (clients) were then distributed across multiple *mI. large* instances as necessary. Figure 5. shows the average message latency versus the number of clients. Note: an average message latency of 300 milliseconds or less is required for real-time video conferencing applications [27].
Workaround for combined images.Eg.- 1000116 Fig. 5

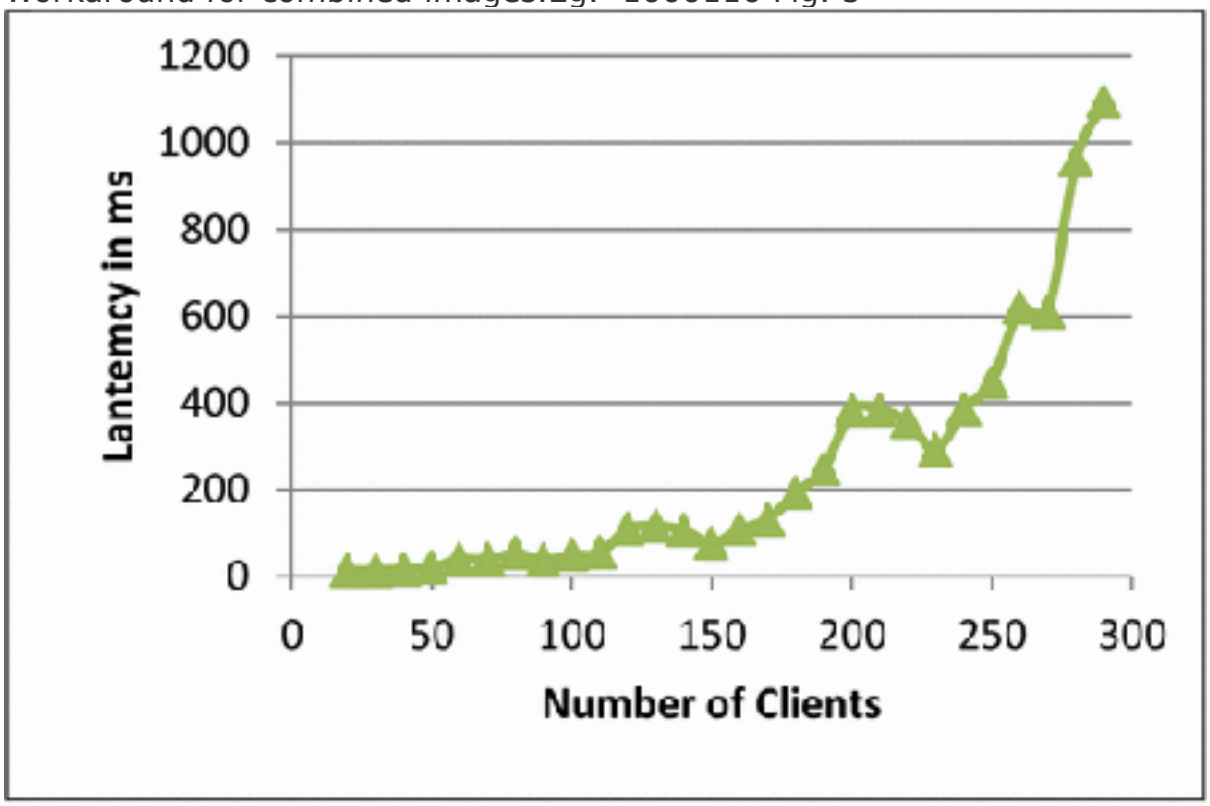


Figure 5. Single Broker Average Message Latency

View All

Therefore if we only consider message delivery times we would conclude that single broker is capable of supporting approximately 200 clients participating in a simulated video conferencing application. However, in real-time collaborative video applications message latency is not the only factor; uniformity of the message latency must also be considered. In order to achieve a satisfactory user experience the video packets must also be delivered in a

uniform (i.e. non-jittery) manner [29]. Figure 6. shows the average jitter versus number of clients.
Workaround for combined images.Eg.- 1000116 Fig. 5

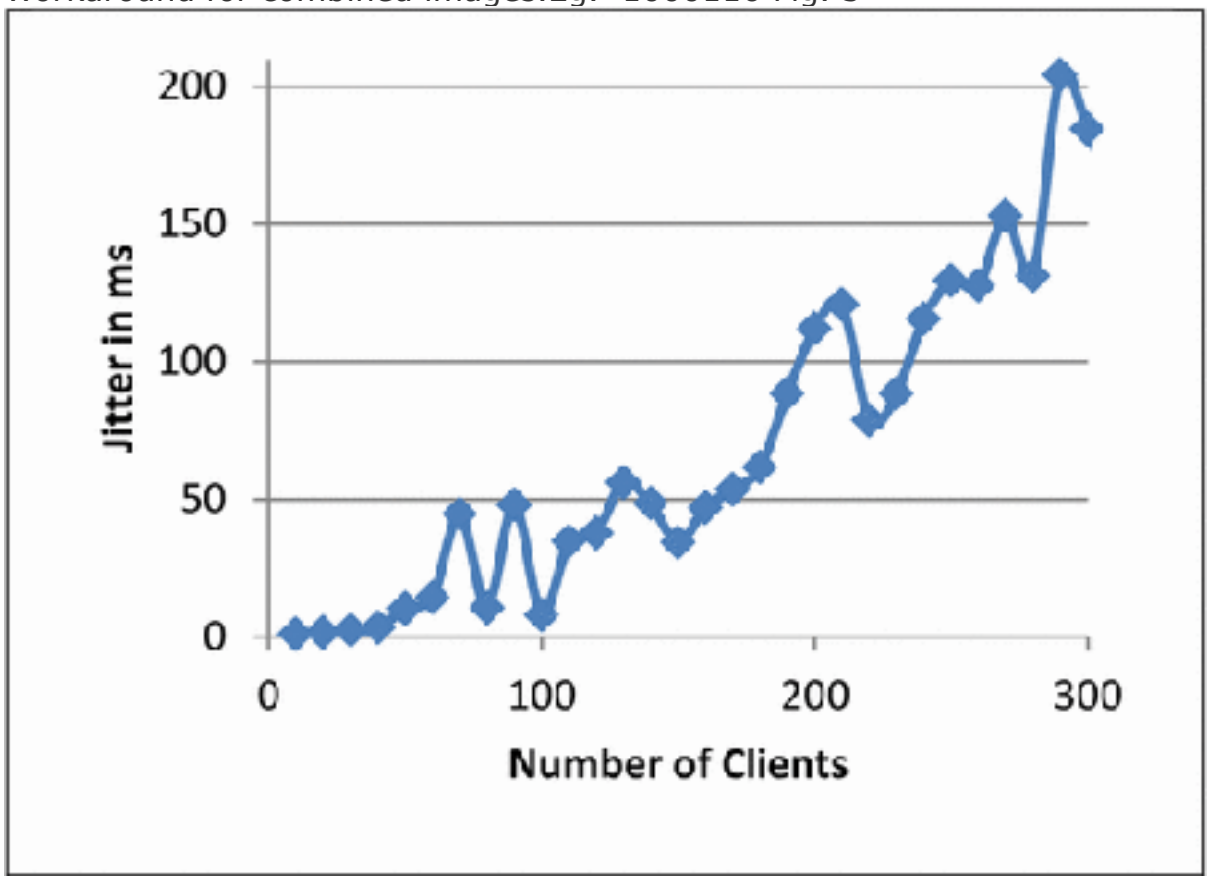


Figure 6. Single Broker Average Jitter

View All

Here we see acceptable jitter until we reach approximately 150 clients. This figure is a better estimate of the true number of clients a single broker can effectively support. We test this conclusion we also examining the jitter as a function of time (packet number). These results are show in Figure 7.
Workaround for combined images.Eg.- 1000116 Fig. 5

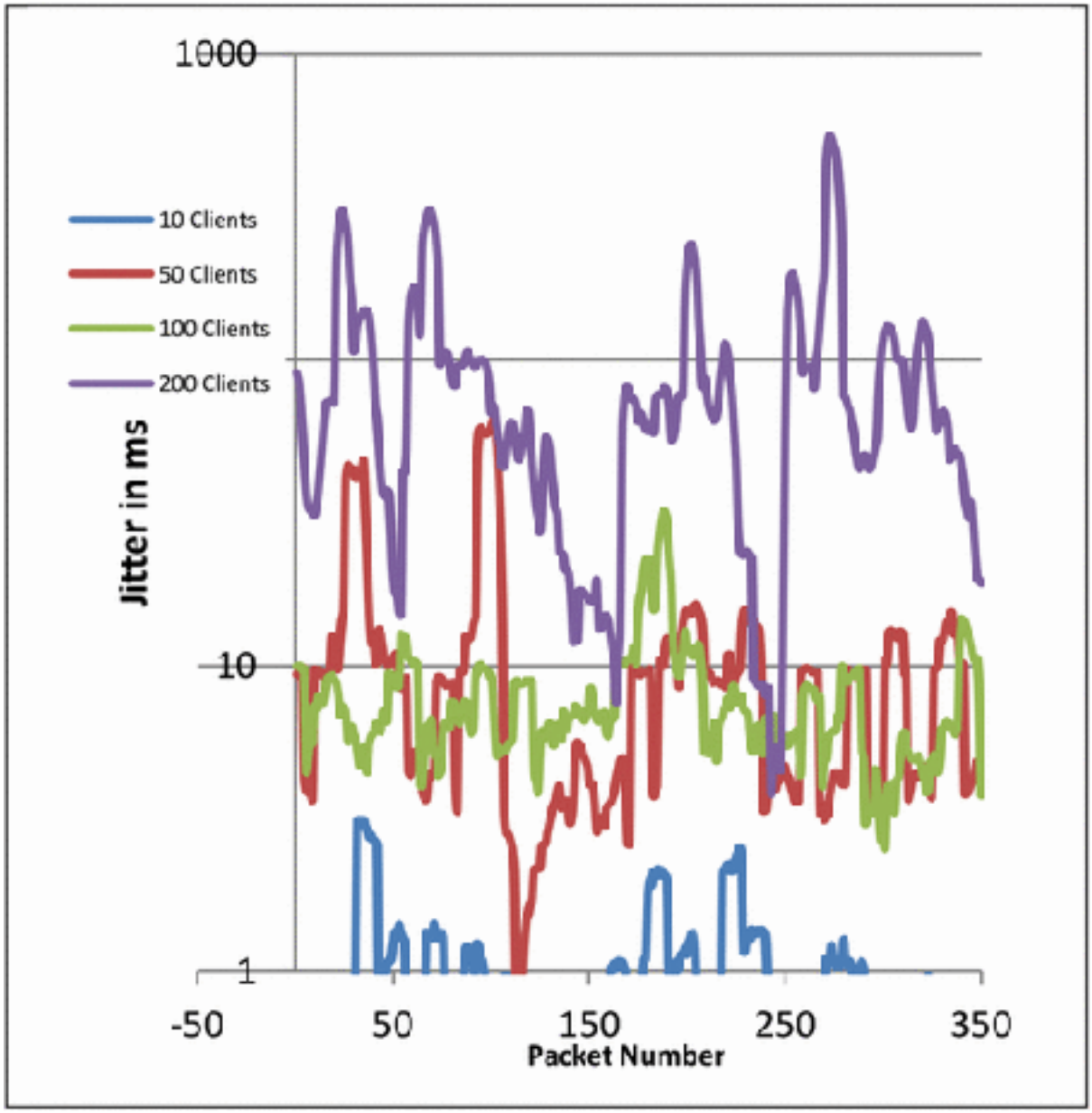


Figure 7. Jitter versus Packet Number (Time)

View All

Again the jitter is acceptably low until the number of clients reaches approximately 150 clients. We have demonstrated, therefore, that a single broker is capable of supporting 150 clients participating in a real-time video conferencing application. (Where 640×480 video is streamed at 30 frames per second.) Scaling may be achieved by deploying additional brokers to Support Iarger client loads.
In previous work [27][28][29] we have shown a single broker is capable of supporting a greater number of clients in cases of: smaller date packets or lower publication frequencies. Many loT applications will consist of sensors utilizing significantly smaller

packet sizes and lower transmission rates - e.g. GPS, RFID, or ZigBee devices - and in these applications we expect our system to support potentially thousands of clients with a single broker. Further work is planned to examine these scenarios.

SECTION VI.

Conclusion

We presented an overview of our open source loT Cloud framework, it's API for creating scalable loT applications, and included examples of currently implemented sensors modules. Next we described the experimental test-bed for cloud development used in our research. Finally we conducted a preliminary study to analyze the performance characteristics of our middleware in the context of high end real-time video sensors. Finally we encourage you to try out the IoTCloud software for yourself. If you are interested please see our site: https://sites.google.comlsitelopensourceiotcloud
Also if you are interested in the FutureGrid you may apply for a free account at: https://portal.futuregrid.org/