H-Plane: Intelligent Data Management for Mobile Healthcare Applications

Rahul Krishnan Pathinarupothi[1(✉)], Bithin Alangot[2], Maneesha Vinodini Ramesh[1], Krishnashree Achuthan[2], and P. Venkat Rangan[1]

[1] Amrita Center for Wireless Networks and Applications (AmritaWNA), Amrita School of Engineering, Amritapuri, Amrita Vishwa Vidyapeetham Amrita University, Clappana, India
pprahul@gmail.com

[2] Amrita Center for Cybersecurity Systems and Networks, Amrita School of Engineering, Amritapuri, Amrita Vishwa Vidyapeetham Amrita University, Clappana, India
https://www.amrita.edu

Abstract. We present an intelligent data management framework that can facilitate development of highly scalable and mobile healthcare applications for remote monitoring of patients. This is achieved through the use of a global log data abstraction that leverages the storage and processing capabilities of the edge devices and the cloud in a seamless manner. In existing log based storage systems, data is read as fixed size chunks from the cloud to enhance performance. However, in healthcare applications, where the data access pattern of the end users differ widely, this approach leads to unnecessary storage and cost overheads. To overcome these, we propose dynamic log chunking. The experimental results, comparing existing fixed chunking against the H-Plane model, show 13 %–19 % savings in network bandwidth as well as cost while fetching the data from the cloud.

Keywords: Cloud computing · Healthcare IoT framework · Log storage system

1 Introduction

Remote monitoring of patients through the use of wearable sensors and smartphones is becoming an effective tool for quality healthcare delivery. If scaled up, it can reduce the load on hospitals as well as the need for patients to visit the hospitals multiple times.

Technologies such as low cost body attached sensors, powerful smartphones which can act as Internet of Things (IoT) gateways for these sensors, cheap and easily deployable cloud solutions and many other innovations have become enablers for remote monitoring applications. Most of the current applications leverage cloud for storage and analytics of data. The cloud also acts as an entity, which helps the globally distributed and highly mobile IoT devices to interconnect. However, as the edge devices (such as IoT gateways and smartphones) are

M. Younas et al. (Eds.): MobiWIS 2016, LNCS 9847, pp. 283–294, 2016.
DOI: 10.1007/978-3-319-44215-0_23

becoming powerful, the cloud is also moving closer to the edge as discussed in [1]. This has helped to build IoT infrastructure which delivers low latency response for applications and reduce network cost of sending large amount of unfiltered data to the cloud.

Based on these developments, we present a 3-tier architecture for remote healthcare applications. We call this architecture as H-Plane, which stands for Healthcare-Plane. It consists of a modified log abstraction for data management that can facilitate location unaware routing and scalable storage. We also present a user access pattern based log prefetch model that can highly improve the efficiency of data reads from the cloud, compared to other systems such as Bolt [2] and Global Data Plane (GDP) [3].

In H-Plane, the log is the fundamental storage abstraction for transferring and managing data. The logs are divided into segments and distributed across different nodes to meet quality of service (QoS) needs. Even though the logs are divided into segments and distributed, the applications have a single logical view of Log. The segments are again divided into logical chunks. A chunk constitutes a contiguous sequence of records and it is the basic unit of data access. Accessing data from the cloud as chunks helps to prefetch data, which in turn reduces the number of requests and round trip delay [2]. However, the method of fixed size chunking in GDP, Bolt and other log based systems, leads to unnecessary storage and cost overheads due to unused prefetch data. In order to overcome this, we present a method for dynamic chunking of the logs based on the user access patterns.

The rest of the paper is arranged in the following way. Section 2 describes the background and related work. The H-Plane architecture, log data abstraction and different log operations are discussed in Sect. 3. We introduce dynamic chunking models in Sect. 4, followed by the evaluation and cost benefits in Sect. 5. The future direction of the research and conlusion is presented in Sect. 6.

2 Related Work

Earlier IoT and cloud architectures as summarized in [4–6] considered the IoT devices connected to each other through the cloud. As the edge devices gained more capabilities, the concepts such as Fog computing [1] gained prominence, which enabled the use of edge devices as storage and processing nodes. As IoT and cloud infrastructures are increasingly used for various applications, scalability becomes a big factor. It is argued in the work of Zhang et al. [3] that the current architecture would not scale to the needs of future applications and hence a data level abstraction, including a log data structure was suggested. The need for a log based storage for numeric time series data such as EEG and ECG is also well studied by Shafer et al. [7]. One of the research work by Gupta et al. [2] leveraged these ideas and implemented the same for connected home devices. Global Data Plane [3] was proposed to overcome the challenges in Bolt [2]. It utilizes the data management capabilities of Bolt such as time-series append-only log, chunking of logs for performance, policy-based storage, data confidentiality and integrity. In order to meet the scalability needs, they have used OceanStore [8], which is a highly scalable distributed storage system. To provide location independent routing, concepts from Named Data Network (NDN) [9] were also incorporated in GDP. Since this is a new concept, we do not know of any applications that have used or tested this framework. Our previous experience [10] with remote monitoring healthcare applications provided the impetus to adapt GDP like architectures for healthcare applications. We feel that it is going to highly simplify the application development and at the same time ensure data integrity.

Many of the existing healthcare applications have been using various cloud and IoT architectures. As the ground realities are changing, the applications need to move towards a more scalable architecture, that is both secure and at the same time ensures certain QoS that are specific to healthcare applications. In this paper, we have enhanced the data management capabilities of healthcare applications using H-Plane. Though it is only in the early stages of conceptualization, we have built upon GDP to design H-Plane as a framework for deploying healthcare applications.

3 H-Plane

3.1 Architecture

In this section, we describe the architecture for H-plane and explain how we have tailored data management for healthcare applications. We consider an extended cloud architecture where there are various body sensors connected to the patient. Usually, these sensors send the data to an IoT gateway, which could be the patient's smartphone or another kind of high-end device. These high-end devices have the capability to perform minimal computation and real-time processing. There could be multiple high-end devices which are in the neighbourhood. These high-end devices could have varying capabilities and could be linked over a heterogeneous network, such as WiFi or Bluetooth. It may be noted that the patient's high-end device could also be connected to another patient's device or to a health service personnel (HSP), such as that of a doctor or a clinician. The high-end devices are connected to the cloud (private or public) over different media and networks. The cloud has higher capabilities for batch processing as well as large storage space for long term archival of data. The public and private parts of the cloud are also connected through proper interfaces. In the following part of the paper, the sensors, the devices and the cloud are referred to as nodes in general. This is depicted in Fig. 1.

Here, the edge devices can communicate with each other and share their processing and storage capabilities, independent of the cloud. The patient's data could be routed to the doctors high-end device skipping the cloud, thereby reducing the upstream traffic to the cloud. The data archival in the cloud could be done at a later time when the cost of transmission is less. This architecture allows the high-end devices to go offline from the cloud and then join in later through a different network. The connectivity of the devices and location unaware routing are managed using NDN [9].

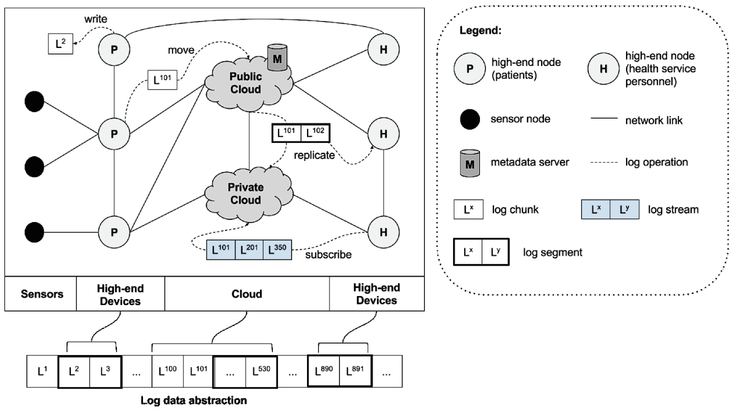

Fig. 1. The H-plane architecture.

Figure 1 also shows a global logical log, which has multiple physical log segments that are located in different nodes. A detailed discussion on logs, chunks, segments and log operations follows in the next section.

3.2 Logs

In H-Plane, a single-writer append-only log is the basic storage abstraction for the healthcare applications. The application gets a single logical view of the log, although it is physically divided into different segments. These segments could be placed at different nodes (cloud, gateways etc.) in the infrastructure to meet various Quality of Service (QoS) requirements as shown in Fig. 2(b). In a log, only the head segment is mutable and all other segments are immutable. The segments are

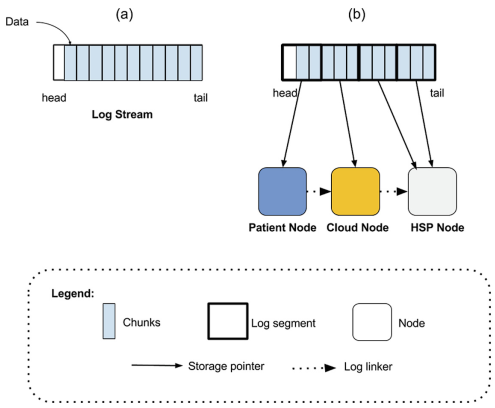

Fig. 2. The H-Plane log abstraction. (a) Data is written to the head of a log segment. (b) Log segments are placed in a distributed manner across different nodes.

divided into different logical chunks. A chunk is the basic unit of data access in our infrastructure and helps in batching data during a read operations which improves the system performance. In the below discussion on logs and log operations, we assume constant chunk size for the sake of clarity. We introduce the concept of dynamically deciding the chunk size based on the readers access pattern, and is explained in a later section. The healthcare applications can use this log abstraction to store and move data between different nodes. H-Plane provides few basic operations to enable applications to effectively use log data structure. These may be further extended based on future needs.

3.3 Log Operations

We define a set of basic log operations that can be used by the applications using API function calls to the H-Plane.

- **Log Creation.** An application can create a new log by calling the following API functions:

  create (user_id, sensor_id=none, sensor_type=none, segment_size=default)

  Using the above, a new log stream is created and a locally generated log_id is returned. The log_id is a unique identifier assigned to a log from a 256-bit address space. The details of the newly created log is then sent to the metadata server. The segment_size can either default to a preset value or could be assigned according to the type of sensor used. This flexibility would allow in optimizing performance of the log management for different kinds of sensors.
- **Log Write.** The medical data received from the sensors are appended into a log in the form of data record; a time-tag-value pair, which may be changed according to the requirements of the application. The append() API function is provided to write data to the log. This API function is initiated with the following parameters:

  append (log_id, value, tag, timestamp)

  The append operation identifies the mutable segment corresponding to the log and then adds the new data record to its head. Once the append operation is complete, it returns the file offset of the data record. The offset is added to the segment's chunk index which is stored along with the segment.
- **Log Read.** A single log has multiple readers, which enables data sharing amongst different applications or users. A user is able to retrieve the data within a particular timeframe using the read API function. It has the following parameters:

  read (patient_id, log_id, start_time, end_time)

  To retrieve data record from the log for a specific time window, the application identifies the location of the segments, which is stored in the segment index (SegmentIdx) in the metadata server. It has the details such as the location, segment_id and time_window. After identifying the segment location, it downloads the corresponding chunk index (ChunkIdx) from the remote device that has the segment. The chunk index contains entry for each data item in the segment. The application queries the chunk index locally to identify the offsets related to the chunk. It then sends a request to the remote device for the chunk and retrieves the data.

We present a special use case that makes use of logs extensively. Suppose a user, such as an HSP or patient needs to retrieve a part of the data. The application can request for log chunks, either from the same log stream or from different log streams. For example, if the doctors want to see all the sensor readings from a patient during a time frame, then the system can retrieve the log chunks from different sensors, which have those timestamps. Another utility is that we can create a critical log stream, that contains only the log chunks which are flagged as critical. This will help in better management of critical data across the cloud infrastructure.

- **Log Subscribe.** The users can subscribe to a log stream to get instant updates from a particular IoT device. When an application calls a subscribe API, the corresponding user id is added to the subscribers list associated with a log stream, which resides in the metadata server. This list is updated when the user moves to another location or when he switches his device. The following parameters are passed to the subscribe API:
  subscribe (log_id)
  Once a user is added to the subscriber's list, the mutable segment of the log is asynchronously replicated to the node associated with the user. A cloud backup node can be made, by default, a subscriber to all the log streams. If a subscriber node goes offline, the data from the source node need to be temporarily stored and sent at a later time, when the node restores the network connection. This storage may be done locally or in the cloud or in the neighbouring nodes.
- **Log Replication.** The reliability and availability of medical data is of utmost priority. Hence, we can make use of the storage capabilities of the neighbouring nodes as well as the cloud, viewing them as a shared storage space. A log segment can be replicated by using the following API call:
  replicate (log_id, segment_id)
  This will inturn send the segment to all the subscribers as well as the neighbouring nodes. The number of neighbouring nodes, time of replication and medium selection are all chosen by the application according to the data management policy as well as the user preferences. In case of replicating a mutable segment, the application will need to send the data records to the subscribers and neighbouring nodes as and when the data is written.

As seen above, the log abstraction model forces the applications to access the data as chunks. Since the log data is time series in nature and the users generally tend to access near by data, chunking is an effective method to prefetch data from the cloud. However, in most of the healthcare applications, the end users are of different categories. For instance, the doctors, technicians, clinical assistants and patients access the log in different ways. Also, the temporal variability in access patterns can also be seen even among the same user group. Suppose the doctor requests a one minute ECG data for a particular patient. The downloaded chunk contains two minutes of extra data which may not be accessed later. It would be inefficient in downloading that chunk into his smartphone. However, in case of a technician, he might use all the three minutes of data. Thus, a constant chunk size leads to inefficient prefetch of data from the cloud, resulting in cost and bandwidth overheads.

We identified that the log writer cannot decide the chunking size since the log reader preferences are varying even on the same segment of data. In order to overcome this challenge, we propose a dynamic log chunking model based on the reader access pattern.

4 Dynamic Log Chunking Model

In this section, we elaborate our approach towards chunking the log segment based on user or reader access patterns. As shown in the Fig. 3, a log chunk, requested by a reader, consists of two parts: requested data - R_{size} (data that is requested for current use by the reader) and extra data - R_{ed} (data that is prefetched along with the requested data). Extra data can be classified as unused data - D_{un} (extra data that is not used by the reader for succeeding requests even though it is stored locally) and used data - D_u (extra data that is used in the succeeding requests). Our aim is to reduce the unused data by dynamically deciding the chunk size for each reader based on access pattern. In order to achieve this, we analyze the amount of D_{un} and D_u for each chunk request. The relationship between D_{un} and D_u is a direct measure of the readers access pattern, and hence we can change the chunk size C_{size}, for the next request based on these. Suppose C^n_{size} is the current chunk size and C^{n+1}_{size} is the chunk size for the $(n+1)^{th}$ request, then we formulate it as:

$$\boldsymbol{C^{n+1}_{size} = C^n_{size} - D^n_{un}}, \;\; if\ D^n_{un} > 0. \tag{1}$$

$$\boldsymbol{C^{n+1}_{size} = C^n_{size} + \alpha * D^n_u}, \;\; if\ D^n_{un} = 0\ and\ D^n_u > 0. \tag{2}$$

$$\boldsymbol{C^{n+1}_{size} = C^n_{size} + \beta * R^n_{size}}, \;\; if\ D^n_{un} = 0\ and\ D^n_u = 0. \tag{3}$$

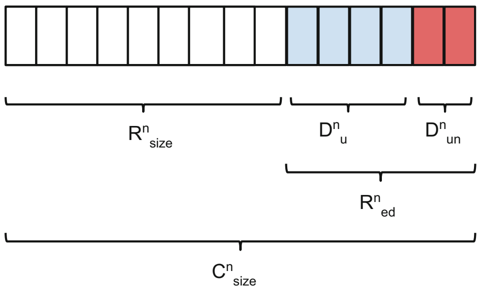

Fig. 3. Representation of a log chunk showing the requested, extra, used and unused data sizes.

Equation (1) is used when there is unused data in the current chunk and it effectively reduces the unused data from the succeeding chunk size. Equation (2) is used when there is no unused data in the current request. This implies that all the prefetch data was useful. Hence, we can slowly increase the prefetch size by using a growth rate α. The value of α is decided by the application to control the growth rate of chunk size. Equation (3) is used when both the used and the unused data in the chunk is zero. It means that the request size was equal to the chunk size, thereby leaving no space for any prefetch data. Hence, we can increase the chunk size by a factor β of the current request size. Once again, the value of β is decided by the application based on how it wants to control the growth rate of chunks. The update of the chunk size could be done after each request or after multiple requests. In H-Plane we have used two different models to update the chunk size.

- **Dynamic Chunking 1 (DC 1).** The Eqs. (1) to (3) are applied based on the average of D_{un}, D_u and R_{size} over n requests. Accordingly a new chunk size C_{size} is calculated and used for the next n requests.
- **Dynamic Chunking 2 (DC 2).** C_{size} is calculated after each request but not updated until a predefined number of requests, n, are completed. The average of C_{size} over n requests is calculated and then used as the new chunk size for the next n requests.

The frequency of updation of chunk size must be based on the frequency of reader access. If the logs are accessed with high frequency, then we can learn for larger number of requests and then use that. On the other hand, if the reader accesses the logs less frequently, then we may use those few access history to learn and calculate the new chunk size. In both cases, the goal is to learn and calculate new chunk sizes faster. This also implies that the frequency of changing chunk sizes is inversely proportional to the frequency of read requests. Hence, we formulate the update frequency parameterized by the frequency of requests and is written as follows:

$$\boldsymbol{n = \gamma * f.} \tag{4}$$

In Eq. (4), n is the number of requests to be used for learning the new chunk size. The value of γ is used to control the update frequency, and it is up to the application to decide. The frequency of requests on that log from a particular reader is represented by *f*, whose unit is in requests per day. Using this equation, the application can find out the frequency at which the chunk sizes should be updated as well as the number of read requests to calculate the new chunk sizes.

To implement the above said dynamic chunking model, we propose modifications to the data read/write operations and the chunk indexing methodology. The data write should be done at the record level, while the data read at the chunk level. Once a complete segment is written and made immutable, an associated chunk index is created which contains the timestamp and the corresponding offset for each data record. This implies that the chunk index will contain as many entries as the number of data records in that segment. When a reader requests for data from a segment, the application downloads this chunk index locally on that device. For every request of size R_{size}, the corresponding offsets are identified from the chunk index. Based on the current chunk size C_{size}, the prefetch size is calculated and an offset is identified accordingly. The final request will thus have the requested data as well as the prefetch data size added together. The remote device, which has the segment, will retrieve the data according to the C_{size} and send it back. It may be noted that for the first read request on a log, a default chunk size may be used.

5 Evaluation

To evaluate the performance of the proposed model, we compared the D_{un} that gets downloaded while using dynamic chunking versus fixed size chunking models. We used 384 KB for fixed chunk size. This is equivalent to about two minutes of standard 3-channel ECG data. The request sizes R_{size} was picked up from normally distributed data request sizes, with mean as 230 KB and standard deviation of 40 KB. The used prefetch data D_u also varied with a mean of 60 % of the extra data R_{ed} and standard deviation of 10 %. A total of 100 requests, totalling to around 35 MB of data (equivalent to four hours of three channel ECG data) were considered for the evaluation.

The frequency of updation of chunk size, for dynamic chunking, was fixed at n = 10. This was done based on a sample usage scenario. Suppose there are 100 request/day, we considered $\gamma = 0.1$. Hence, the latest ten requests are used to calculate the log chunk size. After every ten requests, the chunk size was updated based on both the updation methodologies listed in the above section. The values used for the growth rate α was 0.3 and that of β was 0.1. Though we experimented with these values, different values of α, β and γ could be used based on the application requirements. Figure 4 shows R_{ed}, D_u and D_{un} as a percentage of the chunk size while using the fixed chunking method and two approaches of the dynamic chunking model. In both models the chunk size was updated after every

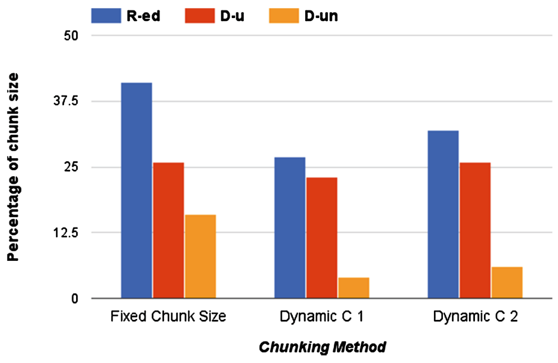

Fig. 4. Comparison of percentage of used data, unused data, and the extra data while using dynamic chunking and fixed chunking models.

10 requests. On an average about 15 % of all the chunk data that is prefetched is not getting used in subsequent requests (as expected according to our mathematical modelling of used data sizes). However, in such cases, the dynamic chunking method 1 decreases the unused data to less than 5 %. Along with this decrease, a slight decrease in the used data is also seen. Using the second dynamic chunking model, the unused data percentage stays at 6 % while the used data percentage is same as that of the fixed chunking method. This reduction in the unused data without affecting the percentage of used prefetch data is considerable and can lead to savings in both costs and bandwidth for the service providers as well as the end user.

5.1 Benefits

Bandwidth Benefits. We notice that on an average about 15 % of all the chunk data that is prefetched is not getting used in subsequent requests (as expected according to our mathematical modelling of used data sizes). However, in such cases, when we use the dynamic chunking method 1, the unused data percentage decreases to less than 5 %. Along with this decrease, a slight decrease in the used data is also seen. Using the second dynamic chunking model, the unused data percentage stays at 6 % while the used data percentage is same as that of the fixed chunking method. This reduction in the unused data without affecting the percentage of used prefetch data is considerable and can lead to savings in both costs and bandwidth for the service providers as well as the end user. The fact that DC 2 has relatively higher D_u also suggests that the reduction in chunk size has not reduced the used prefetch data in absolute terms.

Cost Benefits. The reduction of unused prefetch data also reduces the overall data download and upload requirements, both at the cloud as well as at the end user nodes. Table 1 compares the data downloaded from 100 requests for fixed and dynamic chunking models, given the same request size. DC 1 gives around 20 % savings while DC 2 saves around 14 % in comparison to fixed chunking.

These data saving translates to similar cost savings too. For instance, an application provider using the Amazon S3 for data storage would be charged around \$0.09/GB for outbound data bandwidth for up to 1TB per month. A reduction of 14 % in out-bound traffic would result in savings of \$126 per month. On the other hand, for the end user who uses mobile data for accessing patient data, a savings of 14 % translates to around \$10/GB. Apart from

Table 1. Data usage comparison (in KB) using dynamic chunking and fixed chunking

	R_{size}	R_{ed}	Total data	Savings %
Fixed chunk	22,546	15,854	38,400	-
Dynamic chunking 1	22,546	8,304	30,850	19.7
Dynamic chunking 2	22,546	10,584	33,130	13.7

the cost perspective, the dynamic chunking could result in much efficient use of bandwidth across the entire IoT infrastructure as well.

6 Conclusion and Future Work

The H-Plane architecture for remote monitoring healthcare applications provides a data centric abstraction using logs and related log operations, thereby viewing the entire IoT infrastructure including the cloud, the edge devices and the sensors as a single storage, processing and routing infrastructure. Our experience with using fixed chunking of logs in healthcare applications presented a particular problem of unnecessary prefetch from the cloud and remote devices. The proposed solution was found to improve the performance of log systems by around 15 % translating into cost and bandwidth savings for the cloud user as well as the end users. We envisage that the use of dynamic chunking would be explored further in other domains as well and that other models would be developed that could further enhance the performance.

Acknowledgments. We deeply appreciate the help of our colleagues and friends who reviewed the paper. A special mention to Uma Gopalakrishnan who provided important suggestions for this work. We are grateful to the Chancellor of Amrita University, Sri Mata Amritanandamayi Devi, for constant support and encouragement in conducting research that has direct societal impact.

References

1. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proceedings of the first edition of the MCC Workshop on Mobile Cloud Computing, pp. 13–16. ACM (2012)
2. Gupta, T., Singh, R.P., Phanishayee, A., Jung, J., Mahajan, R.: Bolt: data management for connected homes. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, pp. 243–256. USENIX Association (2014)
3. Zhang, B., Mor, N., Kolb, J., Chan, D.S., Lutz, K., Allman, E., Wawrzynek, J., Lee, E., Kubiatowicz, J.: The cloud is not enough: saving IoT from the cloud. In: 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2015) (2015)
4. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): a vision, architectural elements, and future directions. Future Gener. Comput. Syst. **29**(7), 1645–1660 (2013)
5. Dinh, H.T., Lee, C., Niyato, D., Wang, P.: A survey of mobile cloud computing: architecture, applications, and approaches. Wirel. Commun. Mob. comput. **13**(18), 1587–1611 (2013). Wiley Online Library
6. Bui, N., Zorzi, M.: Health care applications: a solution based on the internet of things. In: Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies, pp. 131. ACM (2011)
7. Shafer, I., Sambasivan, R.R., Rowe, A., Ganger, G.R.: Specialized storage for big numeric time series. In: 5th USENIX Workshop on Hot Topics in Storage and File Systems (2013)

8. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C.: Oceanstore: an architecture for global-scale persistent storage. ACM Sigplan Not. **35**(11), 190–201 (2000)
9. Zhang, L., Afanasyev, A., Burke, J., Jacobson, V., Crowley, P., Papadopoulos, C., Wang, L., Zhang, B.: Named data networking. ACM SIGCOMM Comput. Commun. Rev. **44**(3), 66–73 (2014)
10. Dilraj, N., Rakesh, K., Rahul, K., Maneesha, R.: A low cost remote cardiac monitoring framework for rural regions. In: 5th EAI International Conference on Wireless Mobile Communication and Healthcare - "Transforming healthcare through innovations in mobile and wireless technologies" (MOBIHEALTH). ACM (2015)