ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTA' DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

DEIS
Dipartimento di Elettronica, Informatica e Sistemistica

TESI DI LAUREA

in
Reti di Calcolatori LS

Monitoring per overlay network nel progetto di servizi immersivi

Candidato:
CLAUDIO CAMPEGGI

Relatore:
Chiar.mo Prof. ANTONIO CORRADI

Correlatore:
Chiar.ma Prof.ssa KLARA NAHRSTEDT
University of Illinois at Urbana-Champaign

Anno Accademico 2007/08
III Sessione

On a given day, or in a given circumstance, you think you have a limit.
You go for this limit and you touch this limit and think "OK, this is the limit".
As soon as you touch this limit, something happens and you suddenly can go a little bit further. With your mind power, your determination, your instinct, and the experience as well, you can fly very high.

Ayrton Senna

INDEX TERMS

COMPUTER NETWORKS

DISTRIBUTED SYSTEMS

TELEIMMERSION

OVERLAY NETWORK

QUERY LANGUAGE

PAROLE CHIAVE

RETI DI CALCOLATORI

SISTEMI DISTRIBUITI

TELEIMMERSION

OVERLAY NETWORK

LINGUAGGIO DI QUERY

Table of Contents

1. Introduction

In the last years, the processing power of computers has increased with no limits mainly due to the continuous enhancements of the computational power of the computers. The TEEVE (Tele-immersive Environment for EVErbody) project is an application to take advantage of it, by controlling 3D cameras, a cluster of four 2D cameras, to create a virtual 3D environment. People in different physical locations using TEEVE can join into the same environment and perform a variety of tasks like teleconferencing, remote dancing or telemedicine. TEEVE initially served to experiment collaborative dancing between the two initial TEEVE sites of Urbana, Illinois and Berkeley, California.

At the same time, network connectivity has experienced a blooming period, with the creation, the spreading, and the improvement of many distributed systems, leading to efficient ways to control, organize and manage a large-scale set of nodes with little resources overhead. MON (Management Overlay Network) is a distributed application to monitor a large-scale set of nodes efficiently (i.e., with little overhead) and with fast response time.

Both TEEVE and MON systems are mature, but TEEVE lacks a monitoring system, meaning that all the TEEVE management and deployment tasks are in charge to the controlling person. This challenge motivated the research for the implementation of T-MON, a lightweight distributed monitoring application for the TEEVE system.

The thesis starts with an introduction to distributed systems, then the Chapter 3 describes the TEEVE system architecture. We evaluate some DHT-based structures in Chapter 4, distinguishing between low level solutions, high level solutions, and other DHT-based structures. Chapter 5 provides a detailed description of MON as application and as system. The Chapter 6 illustrates the setup of the T-MON environment and development tools, and Chapter 7 describes the T-MON query

language, from the design process to its final design. Chapter 8 describes the T-MON architecture framework; in Chapter 9 we provide an evaluation and describe the lessons learned.

2. Overview of distributed systems

A distributed system is composed by a group of nodes all connected on a network to perform a common computational job. Each node cooperates to the result of the computation with part of its resources.

Usually each node is a computer but, depending on the abstraction level, it may represent a cluster, where many computers are connected together on a local network. The network connection may be local (LAN), remote (WAN) or there may be a hybrid setup, where group of nodes are connected together on a LAN and to the other groups with a WAN. A WAN (Wide Area Network) connection (also named global connection or Internet) is not guaranteed to be reliable and its characteristics are defined by some metrics: bandwidth, latency, round trip delay, jitter and loss rate are the most used.

Bandwidth is a capacity metric and represents the amount of information (expressed in bits or bytes) that can flow in the link every second while latency is the delay an information takes to reach the destination node once sent by the sender. Round trip delay is the sum of the latency from the sender node to the destination and from the destination to the sender node. Given the best-effort nature of Internet, round trip delay is rarely twice the value of latency time, as each router has different queues for incoming and outgoing traffic. Jitter is the variation of latency over the time, while the loss rate is the amount of information not received by the destination node.

Distributed systems have some widely shared features, like low impact of node failures, some degree of transparency, decoupling the hardware and operating system of the node through common interfaces, scalability and a naming infrastructure.

Being resistant to failures is important, so the impact of a node failure must be the lowest possible on the system, increasing the system availability; almost all modern distributed systems have a failure detection algorithm and/or a

reconfiguration procedure to deal with this situation. When the system's performances are not affected by a single node failure then Independent Failure is achieved. The distributed systems paradigm favors the use of many legacy computers instead of fewer powerful workstations; even if the average failure rate is higher in the legacy computers, the replacement is cheaper than in the all-workstations systems and the aggregate computational power of the distributed system is higher.

Transparency refers to the capacity of the distributed system to hide its complexity to the user. On a completely transparent system is impossible for the user to see if the request is sent to a single computer or to a distributed system. Full transparency has a significant cost and therefore is never implemented in a modern distributed system; the level of transparency for a system is part of the distributed system's design choices and must be carefully evaluated. Higher transparency means a simpler interface to the user and a lower degree of system configuration. For example, a node failure in the system may be either completely hidden to the user or end up in a notification to him or her. Other than the failure transparency just described, other types of transparency are the access transparency, where the access to the system is independent by the type of client, and location transparency, where a naming system is used to allow decoupling between the service requested by the client and the server that will serve the request.

To decouple the functionalities offered by a distributed system from the hardware and the operating system used by the node a common software layer, called middleware, is implemented. The middleware allows the interaction between heterogeneous nodes and provides a common representation for the exchange of data. Given that the middleware provides an abstraction of the layers below it (thus hiding and decoupling them to the above layers), the middleware is also used for local communications at the cost of an overall (but generally small) overhead.

Scalability is the capacity of a distributed system to grow without having a decrease in the overall performances of the system. It is one of the key factors of a distributed system, as normally those systems grow in size with time.

A naming infrastructure allows the communication between nodes of the system and improves the routing of the information from the sender to the recipient, as direct communication between sender and receiver is not the average case; usually the communication passes trough some system nodes. A simple solution is to use the Internet name structure, but many distributed systems use a higher level, dedicated naming architecture.

Each node of the system has its own internal state, but each node stores also its view (that may be global or partial) of the system, like for example information about its neighbor nodes. It is important that the node view is consistent and has not incorrect values as it may reduce the system's performances or break the entire system.

Distributed systems have many metrics and capabilities; their classification depends by which main metrics are used to group them. Distributed systems may have nodes organized in a hierarchical organization or all nodes may be peers. Systems that belongs to the first category are based on a Master/Slave model or are systems where some nodes act as supernodes. Each supernode manages a group of nodes, so that many disjointed groups partition the entire system. When a new node joins the system, a supernode detection and selection algorithm is executed. That algorithm is a design feature of the distributed system; the selection may search for the geographically closer supernode (using for example the subnet name, latency or number of hops) or may randomly select a supernode to increase the strength to system failures. To improve the system efficiency, groups have a similar size and nodes may change group over time because the failure of their supernode or after the join of a node.

Nodes belonging to a distributed system may all provide the same service or each node may provide different functionalities. An example of the former case are the systems based on quick index retrieval, while an example of the latter case are object based midllewares, where each node offer a service described by an appropriate definition language.

Depending on the purpose of the distributed system and when a certain service is available in at least two nodes, a task may be split to all the nodes, that cooperate to achieve a result. Alternatively, the task may be executed on each node to provide a validation of the result against errors, as is the case of mission critical systems. Another alternative is that a task is assigned to one node, allowing different kinds of load balancing.

Node organization is important to decrease communication overhead and to allow a good scalability. The node structure used is typically a tree, a ring, or Directed Acyclic Graph (DAG). Trees and rings are simple structures and require few connections for each node, but are less robust to link failures, so each node has a knowledge of some other nodes that are members of the system and, if a failure happen, the node can contact one of them to rejoin the system.

Data communication between nodes can be synchronous or asynchronous; the key difference between the two is that synchronous communication block the sender's execution flow until completed, while with asynchronous communication the sender continues immediately after it has submitted its message for transmission.

Binary is the format of data communication used for specialized, lower level systems. It is optimized to be lightweight in both payload size and encoding/decoding overhead, but it is not human readable. When the communication involves heterogeneous systems like for example web services, a higher-level language such as XML is used.

3. Description of TEEVE (Teleimmersion) System

TEEVE (Tele-immersive Environment for EVErybody), sometimes also referred to as Teleimmersion, is a system that creates a 3D environment composed by multiple 3D scenes taken at different locations. The two initial sites are located in Urbana, Illinois (at the University of Illinois at Urbana-Champaign) and in Berkeley, California (at the University of California, Berkeley). Each location is equipped with 3D cameras, plasma displays and several powerful computers to manage and process the data in real time [LYY+05].

3.1. Usage scenario

Teleimmersion initial main usage and development scenario is collaborative dancing, with collaborative dancing sessions held between the two sites of Urbana, Illinois and Berkeley, California; another option available is to record a 3D dance session and use it for dancing with a digital alter-ego or for performance rehearsal. This scenario makes a good base to express dancer's creativity. New Teleimmersion-based projects have recently started, including training sessions of a wheelchair basketball team and a multi-site TEEVE capability [WYN+07]. The rendered scene supports visual effects, like a 3D terrain with a mountain and a body of water, some animals and flying stars.

3.2. TEEVE Architecture

TEEVE scene acquisition happens with 3D cameras. The 3D cameras are placed at two different height levels (top and bottom) and arranged all around the location's room, so that there is a complete visual coverage of a person's body, so that a more accurate 3D representation of it can be grabbed. Four 2D cameras compose a 3D

camera cluster; three of them are black and white to get the depth information and the fourth one is a color RGB camera to get the color information.

Each 2D camera is connected to a computer with an IEEE1934 (Firewire) interface; a 3D scene is obtained computing the several inputs taken from the 3D cameras. The synchronization of frame acquisition is extremely important to correctly reconstruct the 3D image, so the frame grabbing process is hardware synchronized.

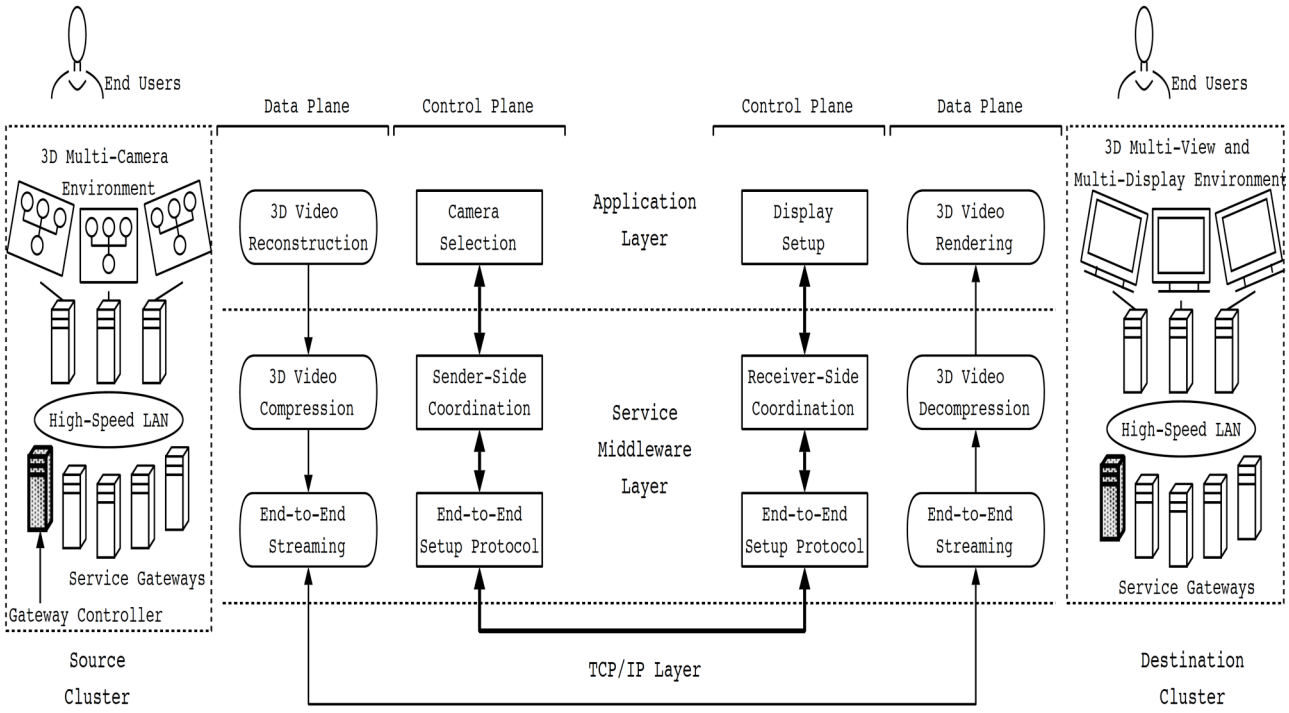


Figure 3.1. TEEVE Architecture

The TEEVE system has three layers, as shown in Figure 3.1: the application layer, the service middleware layer (SML), and the underlying transport layer (i.e., Internet2 [INT96]). The section 3.4. describes the transport layer.

3.2.1. The application layer

The application layer manages the interaction with the end user and the management of cameras and displays' systems: the synchronization of 3D frame acquisition, the reconstruction of a 3D frame, the routing of 3D video streams to multiple displays and rendering the scene from the user-selected viewpoint.

In the data plane it performs a real-time 3D reconstruction and rendering, while in the control plane it gets the user response to control camera selection and display setup. For cross-layer interaction, it needs to inform the lower layer of important system changes.

3.2.2. The service middleware layer (SML)

The service middleware layer provides the needed services to the infrastructure like the 3D video compression, the 3D video stream adaptation, network congestion management and 3D cameras coordination. It is the layer on which the gateway operations are performed: 3D streaming, sender/receiver coordination, and end-to-end feedback control.

3.3. Description of TEEVE nodes

This section describes the characteristics of the TEEVE nodes. Each node is connected to a high-speed LAN for fast data communication. The TEEVE system has the following nodes: trigger, camera, renderer, and gateway.

3.3.1. Trigger

The trigger regularly emits a triggering signal to the 2D cameras on its parallel port pins. The frequency of the trigger signal is about 15 hertz, resulting in a 2D camera frame rate of 15 frames per second. There is one trigger at each TEEVE location.

The trigger receives a notification message every time a camera node grabs a 3D frame.

3.3.2. Camera

The camera is the reference name of a computer connected to a 3D camera cluster that reconstructs the 3D scene from the four 2D streams. The creation of the 3D scene is computationally very intensive, so the processing power of the camera

must be high to be able to get a good frame rate. Every camera computer's metric is CPU bounded, so the overall TEEVE frame rate depends by the output frame rate of cameras; currently the frame rate is between 10 and 11 frames per second. Each camera sends a notification to the trigger every 3D frame grabbed.

3.3.3. Renderer

The renderer displays the 3D scene on one or more display (42" plasma displays are used for TEEVE) to show the output of either the local cameras if the system is working in local mode, or the combination of local and remote 3D environment if the system is working together with a remote location. An alternate reference name for the renderer is display.

3.3.4. Gateway

The gateway is the manager of the system and performs several coordination and management tasks for the Teleimmersion system. The gateway is the only node connected to Internet; all communications between remote TEEVE locations happen only between gateways; when TEEVE is set to work in local mode all the streams are handled from the gateway and then forwarded to the appropriate node. In remote mode the communication is bidirectional, on each direction the gateway aggregates the 3D streams from the local cameras and sends them to the remote gateway, that then route all the 3D streams (local and remote) to the destination local nodes (see Figure 3.2). The complete 3D environment, composed of the union of all the local 3D scenes, is computed and rendered on display computers, so that the overall computational load is evenly distributed among the nodes.

Each TEEVE site is referred to as cluster and is identified by the IP address of the gateway.

In the early versions of TEEVE there were two gateways, the service gateway and the gateway controller, whose duties are now performed by a single gateway node.

3.4. TEEVE communication schemes

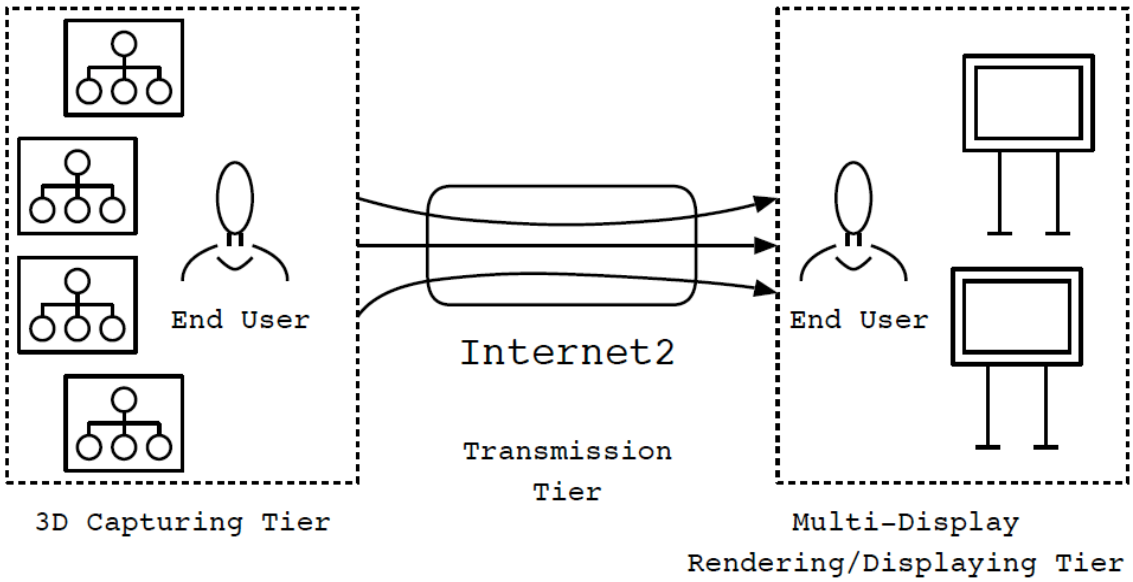


Figure 3.2. TEEVE communication scheme

The Internet2 link between the remote sites of TEEVE, albeit is an high-speed connection, is shared between all the research institutions participating in the Internet2 program, hence it is subject to the common network issues like congestion, round-trip time delay and jitter. The gateway aggregates the 3D video streams together so that in case of congestion it has the control about the action to be taken. Typical actions are the limitation of the output rate of one or many 3D streams, starting from the least important for the scene (the importance of a stream depends on the view selected by the end user), or drop one or many 3D streams, if the congestion is significant. The bandwidth requirement over time of an unmodified 3D stream present spikes corresponding with the output stream's frame rate because of the synchronization requirements, a behavior that greatly increase the occurrence of a network congestion that leads to a decrease of the system's performance. A solution [YCL+05] to this issue consists in distributing the 3D frames grabbed from each 3D camera over the time interval that precedes the following 3D frame, but that solution needs to be carefully tuned to avoid to delay too much the frames. Delaying 3D frames increases the delay perceived by the end user, while the delay is an important

metric for the evaluation of the performance of TEEVE system for collaborative dancing [YYD+06, YYW+06]. On average, TEEVE streams to a remote location require up to three hundred Megabits per second.

Other than the aggregate 3D video streams, each gateway has another connection open for the control channel. The selected transmission protocol for communication between gateways is TCP [YCL+05] due to its reliable and in-order delivery, the congestion control, the easiness of handling large packets at the application layer, and the lower context switching overhead, which are well suited for streaming data of large volume. Although the backoff and retransmission mechanisms seem to make it undesirable for streaming, it has been shown that TCP is widely used in commercial streaming systems.

Being the central node of the system, the gateway has also the knowledge of many TEEVE parameters, such as the number of cameras active, the status of each 2D camera, the bandwidth usage in the LAN and to the Internet2, the network delay, the loss rate and the frame rate of each renderer and of each camera.

3.5. TEEVE operations

This section describes the boot sequence and the monitoring of TEEVE. The boot sequence of TEEVE must follow a defined order: the gateway must be the first node to come online, followed by the trigger and then, independently by the appearing sequence, the cameras and the renderers. The trigger must wait for all cameras to join the system before starting to emit the trigger signal to the 2D cameras. The dynamic join of a camera or a display on TEEVE is allowed, but the local gateway must notify all the other clusters about that; the failure of a node may be tolerated depending on the role of the node in the system. The entire system becomes unusable if the gateway fails or if a camera fails, because the failure of a camera makes the trigger faulty as well. A failure of the renderer, instead, has no

effect on the system other than having one less display available to render the Teleimmersion 3D scene.

3.5.1. System monitoring and maintenance

The entire management of Teleimmersion is mostly manual: most operations are performed from a computer (that isn't part of TEEVE system) opening a Microsoft Windows Remote Desktop Connection application for each node of the system and from there execute the commands or browse the output for a manual check of the correct operation of the node. The deployment of new software versions is very little automated with some scripts, but system monitoring is not. The requirement of a person continuously monitoring the system makes a proactive reaction and adaptation very difficult, and the deployment of new test versions takes time that an automated system may save in favor of research work.

Teleimmersion might be the inspiration for the following future applications: 3D teleconferencing, remote learning (tai-chi, dancing), remote teaching and creativity.

3.6. Multi-site TEEVE

The multi-site TEEVE is an evolution of the TEEVE project to support more than two locations simultaneously [ZY08]. The main issue in this evolution lies in the transmission tier (illustrated in Figure 3.2), because the TEEVE solution does not scale well to many locations. Increasing the number of locations without a new design of the network layer implies that every location is directly connected with each of the other locations, hence the number of connections required increases exponentially with respect to the number of locations. Considering the amount of network resources such as bandwidth required for TEEVE, is clear that the Internet2 network is not able to provide support for the multi-site TEEVE.

The multi-site TEEVE implements two solutions to reduce network load: the support of an overlay network for network path optimization, and the use of stream balancing techniques. The Figure 3.3 shows the architecture of multi-site TEEVE.

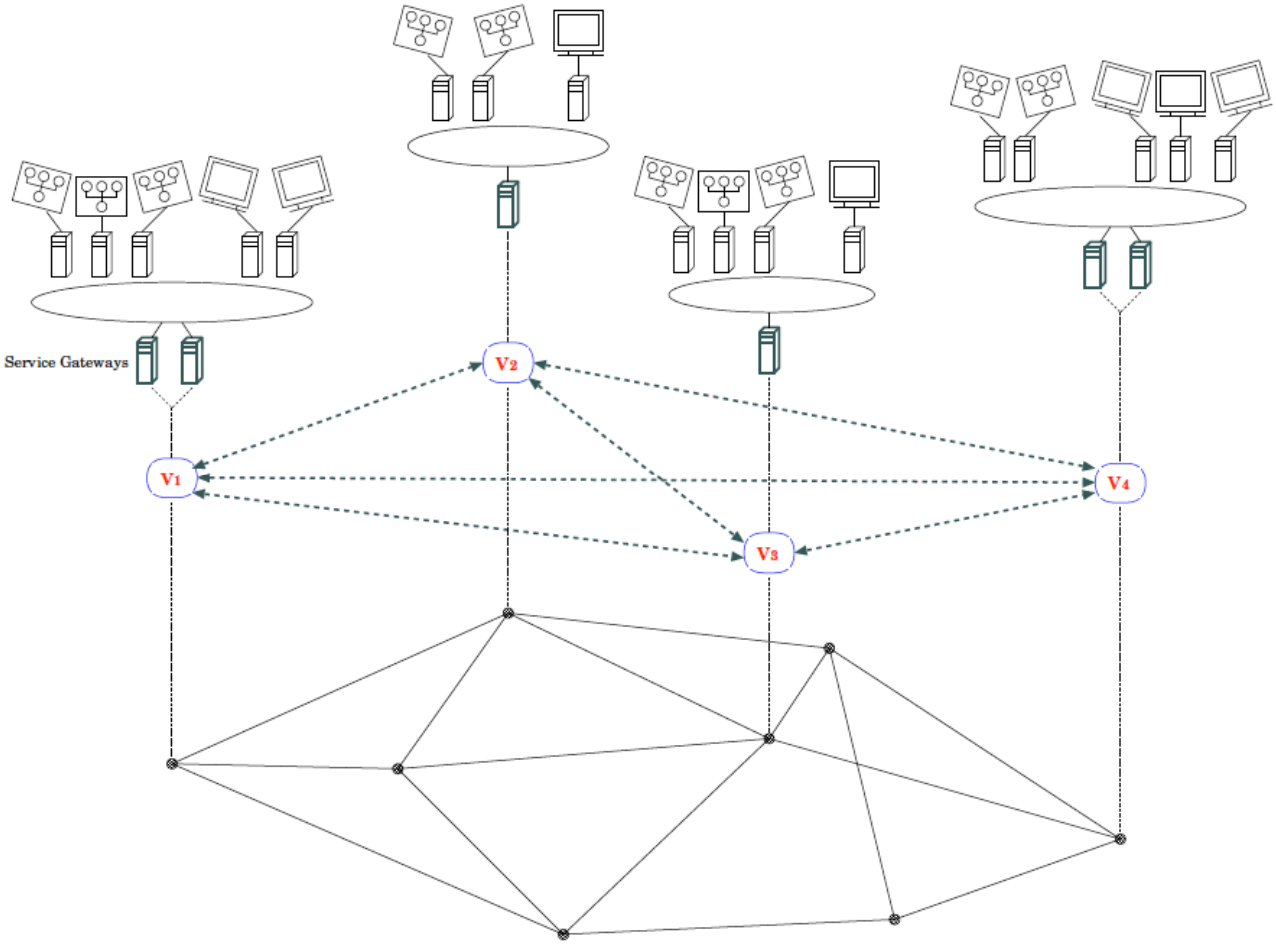


Figure 3.3: Multi-site TEEVE architecture

3.6.1. The overlay network

The overlay network is part of the multi-site TEEVE transport layer and its information are used by the upper layers to optimize the 3D streaming across the overlay. Each gateway is a node of the overlay; in the Figure 3.3 the overlay nodes are the ones labeled v_i. The gateways exchange information to maintain the overlay network and cooperate for content delivery. Some information exchanged by the gateways are the following: overlay network membership, link status, local resources and available streams.

3.6.2. View selection algorithm

The view selection algorithm sends the relevant 3D camera streams to a remote location; a view selection algorithm computes the views to include. It reduces the number of 3D video streams transmitted over the network and provides useful information for the stream balancing techniques.

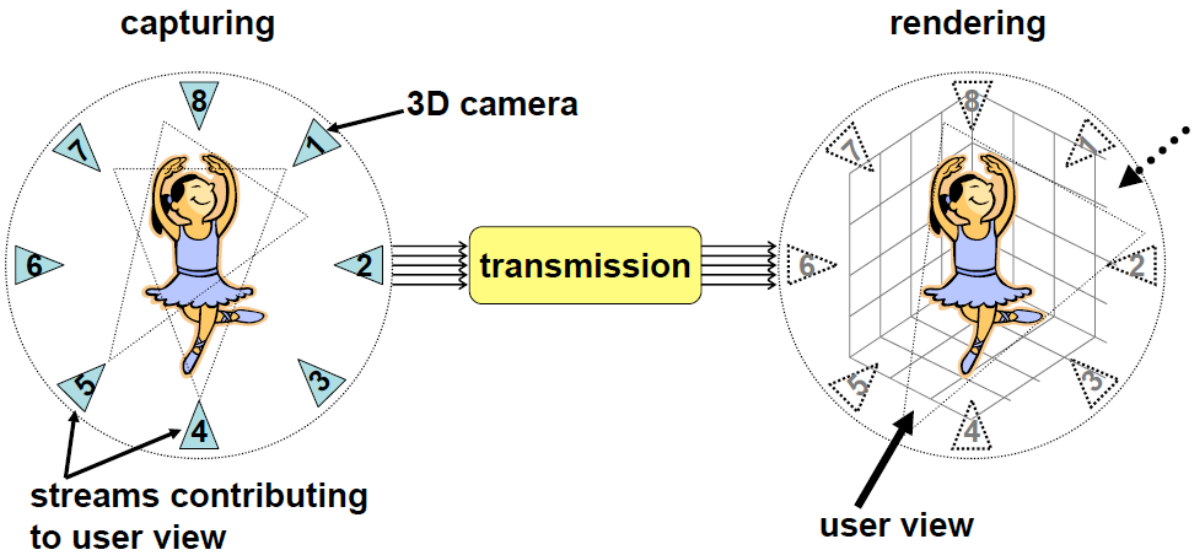


Figure 3.4: The view selection algorithm

Before describing the view selection algorithm, it is important to define a view in 3D space. A view is the visualization point of view of the user. The TEEVE user can arbitrarily rotate and zoom to the 3D scene, creating different views. The view selection algorithm, as illustrated in Figure 3.4, sends to the user only the relevant 3D streams with respect to the user's selected point of view. In the example of Figure 3.4, the cameras 4 and 5, recording the front of the dancer, are the most important, as the overall rendered scene will depend mostly by them. The cameras 1 and 8, that record the rear of the dancer, provide no information to the 3D scene visualized by the user.

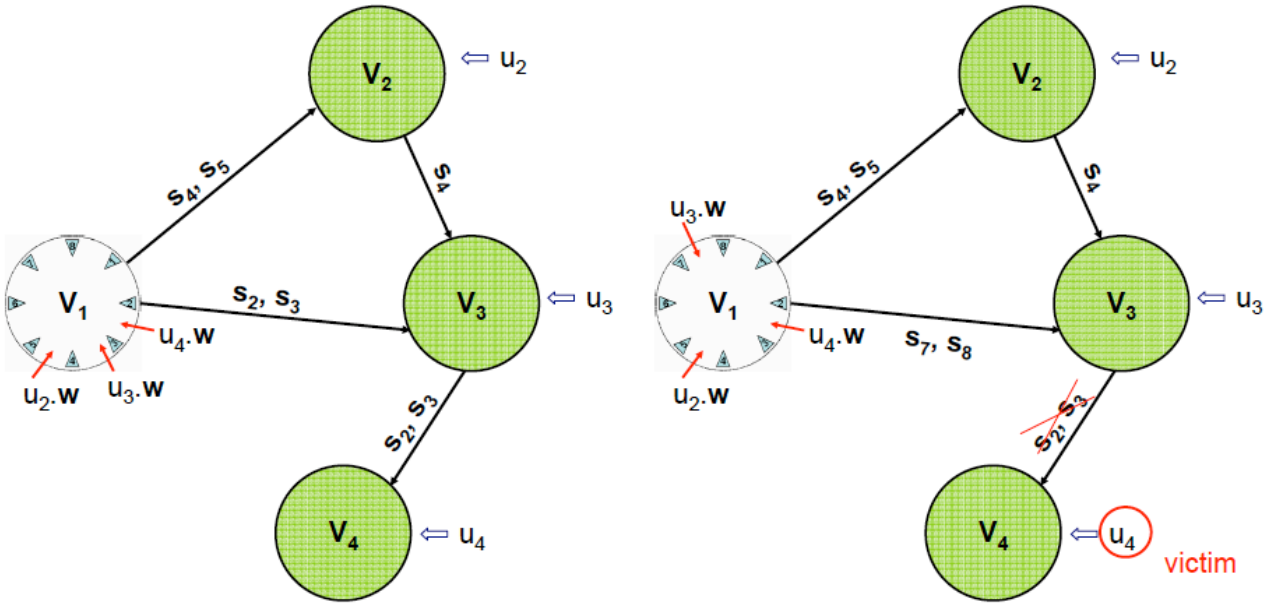
3.6.3. ViewCast and stream prioritization

ViewCast is a view-based content dissemination scheme used for stream coordination in multi-view TEEVE [ZY08]. ViewCast has two major goals: minimum quality guarantee and view change resilience.

The minimum quality guarantee is a coordination between overlay nodes to guarantee that a node receives a minimum set of streams, so that is can have a stream quality guarantee.

The view change resilience coordinates the members of the overlay network so that the change view of one user has a minimum impact on the other nodes. In the example of Figure 3.5 the nodes v_3 and v_4 have a similar view, therefore v_4 receives the streams from v_3. When v_3 changes its view, streams needed by v_4 may become temporarily unavailable. If that happens, v_4 becomes a *victim*.

The strategy applied to improve the resilience of the view change tolerance is the strategy of dependency balancing. The dependency balancing includes three techniques: source balancing, priority balancing and load balancing.



(a) Before View Change (b) After View Change

Figure 3.5: Effect of view change

3.6.3.1. Source balancing

The source balancing (shown in Figure 3.6) tries to evenly distribute the incoming streams of a node across all the node's overlay connections.

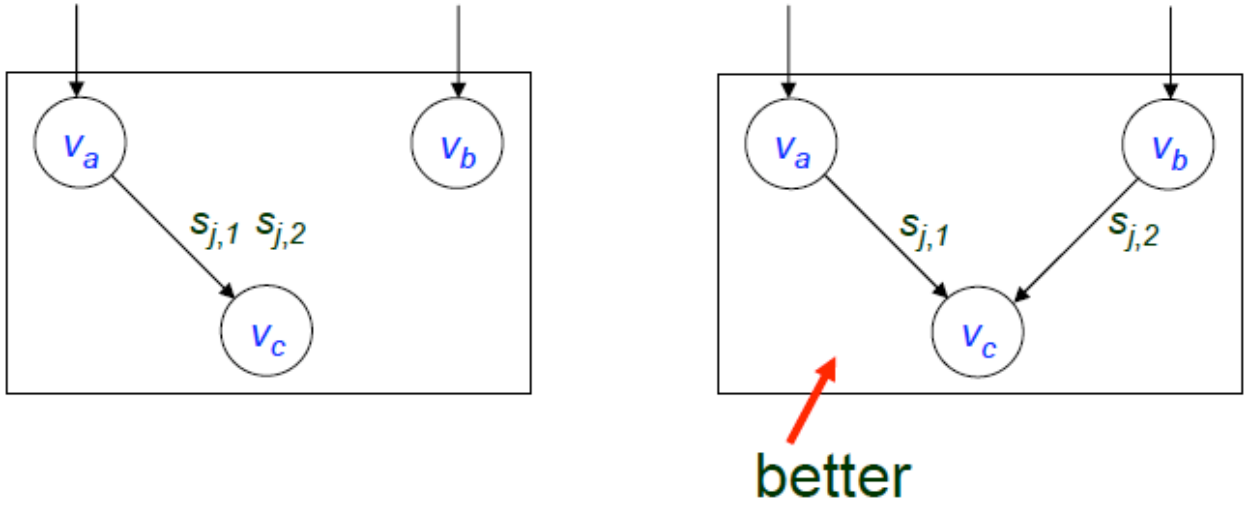


Figure 3.6: Source balancing in ViewCast

3.6.3.2. Priority balancing

The priority balancing in ViewCast (shown in Figure 3.7) tries to evenly spread the average stream priority across all the node's overlay connection. In the example provided in Figure 3.7, the nodes v_a and v_b receive the same stream, but in the left diagram of that figure the two most important streams for v_c, P_3 and P_4, are on the same link. A coordination between v_a and v_b leads to the situation represented on the right section of Figure 3.6.

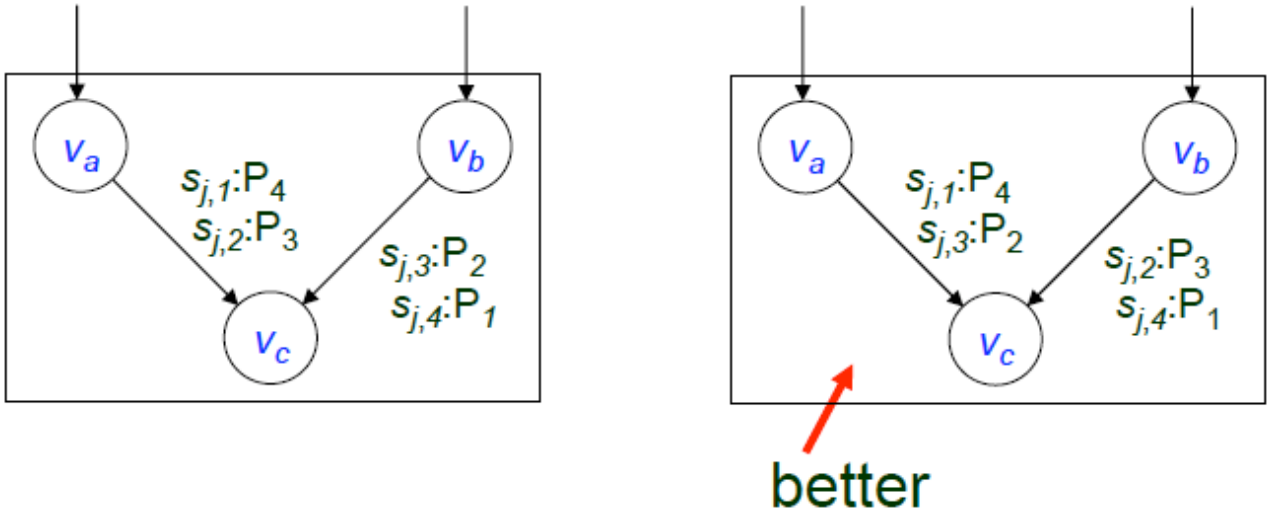


Figure 3.7: Priority balancing in ViewCast

3.6.3.3. Load balancing

Load balancing tries to balance the load among all the overlay nodes.

As shown in figure 3.8, there are two criteria used to calculate the load: total forwarding load and forwarding load to specific vertex. The latter is better because is more balanced, as discussed in the Source balancing section.

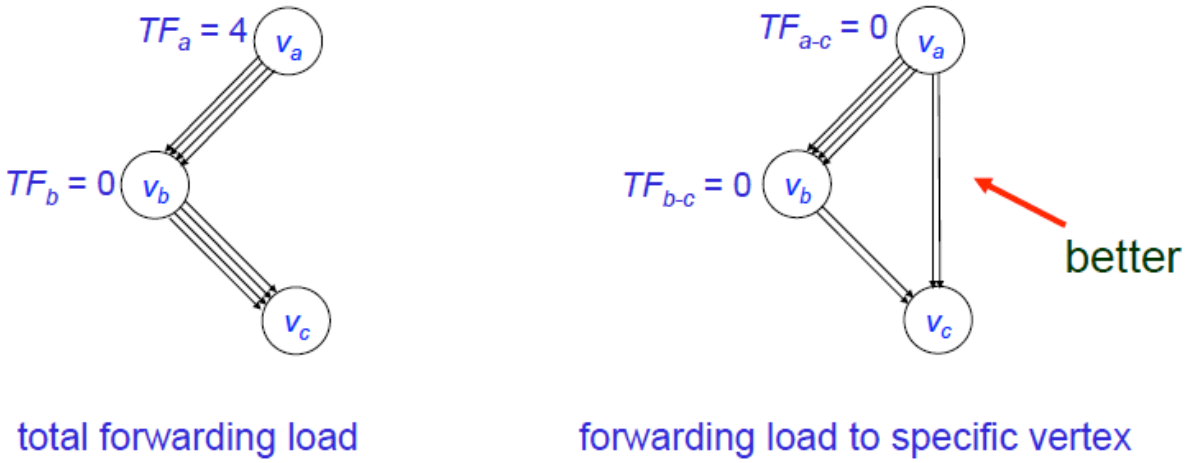


Figure 3.8: Load balancing in ViewCast

4. Related work

This chapter includes the preliminary study of T-MON over TEEVE. The purpose of this work is to explore alternative distributed systems architectures, to see if some of them are more suitable for monitoring of TEEVE than MON. The main concern comes from the fact that MON was designed for large scale distributed systems, while TEEVE can have a hundred nodes at the very best, but on the other hand MON is a tested and reliable solution already available while the possible alternatives have mostly to be coded from scratch.

4.1. Evaluation metrics

The Teleimmersion system has currently a maximum number of one hundred nodes. Even considering the future development of the project, when more than two locations will be able to interact together, it is difficult to imagine a system of thousands of nodes. For this reason scalability was not the main requirement in choosing a solution.

Teleimmersion, instead, is very sensible to latency and partly to bandwidth. Latency is an important parameter and has to be kept as low as possible to have an efficient monitoring system. Bandwidth is not a problem when the query spreads in the same LAN, but is an important factor when the query involves the other locations. Disk space for data storage is not a problem, because currently few information are stored on the nodes. CPU utilization is a very important parameter, because each node must focus on the real-time tasks of video acquisition and elaboration. Other important metrics of DHT-based data structures are: time to insert, time to delete, time to update a node in the system.

The following DHT-based structures belongs to three different groups: the network-level (low-level) solutions, the application-level solutions (high-level), and other DHT-based structures. Sometimes the association of one solution with the group may not be straightforward, so an explanation of the decision will be provided.

Often, it was difficult to evaluate the following DHT-based solutions, as the research papers tend to highlight the strengths of the proposed solution but rarely they list the weaknesses. The weaknesses are therefore guessed from the lack of some evaluation metrics, or from systems comparisons available on other research papers.

4.2. Network-level solutions

This section describes the low-level solutions evaluated: Pastry, Chord and Content Addressable Network (CAN).

4.2.1. Pastry

Pastry is a widely adopted DHT implementation, with an optimistic approach [RD01]. Each node stores the routing table (nodes with similar nodeId), neighborhood table (nodes closest to the current node according to the proximity metric) and a leaf set (list of nodes with numerically closest nodeIds; it increases routing performances). It provides five API calls to easily program it.

The routing is incremental, i.e., at each step the logical hop distance increases. Routing is based on the nodeId of the destination node, so at each step the message passes to the node with most similar nodeId; this allows to perform local routing decisions at each step.

Pastry is fairly resistant to node failures, especially if failures are randomly distributed among the system; routing is not affected by node failures.

Tests shown that routing is optimal with respect to both logical hop counts and real network metrics (like RTT). The simulation ran 100.000 nodes in a simulated network environment; the cost of node join and the number of routing steps grows with O(log N), while there is no information about latency. The throughput achieved is high.

CPU use was higher than other DHT-based implementations, but it is a widely adopted implementation (it is available for the PlanetLab distributed infrastructure).

The optimal routing used on Pastry might provide advantages on the small-scale Teleimmersion network, despite the research paper doesn't disclose information on performances of Pastry network with a smaller amount of nodes. For those reasons, Pastry might be suitable for Teleimmersion needs.

4.2.2. Chord

Chord [SMK+01] is a low-level DHT implementation targeted for frequent node arrivals and departures. It provides just one simple operation: given a key, it maps a key onto a node. Applications can generate a key for each data item and store the data item in the same node where the key is mapped to exploit the efficiency of Chord.

The main characteristics of Chord are the following: it is load balanced, because keys are evenly spread across all nodes; it is fully distributed, because all nodes are peer; and it has a flexible naming, because the Chord key space is flat. Chord hashing function is consistent, so the `keyId` assigned to a node is guaranteed to be unique in the system.

Chord's identifier space has a circular structure, where `successor(key)` denotes a successor of the current node following a clockwise order. Chord preserves two invariants: the first one is that it correctly maintains each node's successor; the second one is that, for every `key`, the node responsible to it is `successor(key)` node.

For efficient routing, each node stores information about O(log N) nodes. Lookup requires O(log N) messages, while node join or leave requires $O(\log^2 N)$ messages. Some applications suitable for Chord are: cooperative mirroring, time-shared storage, distributed indexes, and large-scale combinatorial search.

Simulation and experimental results show that average path length is ($\frac{1}{2} \log_2 N$) and average latency is lower than 285ms with a worst case of 600ms (95% percentile) on less than 200 nodes. With a bigger index table the number of lookup messages decreases.

In the Teleimmersion system there is a strong correlation between nodes (i.e., a failure of one node significantly affects the performances and operability of the entire system). Teleimmersion assumes that its nodes rarely fail, join or leave the network while the system is running. On the contrary, Chord works for distributed systems with frequent node arrival and departures: it means that it addresses a different problem and therefore it is not optimal for Teleimmersion system.

Therefore, since Chord is designed for distributed systems with frequent node arrivals and departures, it addresses a different problem that Teleimmersion needs, being not optimal for Teleimmersion system.

4.2.3. Content-Addressable Network (CAN)

The main feature of CAN is the mapping of nodes in a d-dimensional cartesian space [GS03]; a high value of d means that each node has more neighbors, so the routing distance decreases, but the node resource usage increases as it has to keep track of more neighbor nodes.

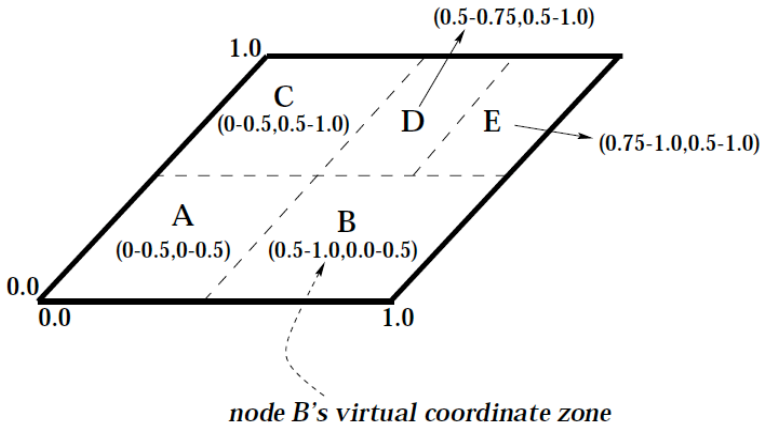


Figure 4.1: Example of a CAN node allocation in a 2-d space

A node join is performed in two phases: first, by selecting randomly a coordinate point P and then splitting the node space in which P lies. Join operation and node failure affect a very limited area of the space. Each node stores 2•d neighbors, while the average path length is $(d/4) \bullet (n^{1/d})$ hops, that is $O(d \bullet n^{1/d})$. Use of flooding may happen only in the extreme case of large simultaneous node failure.

Latency is measured between application level hops, and the CAN's latency optimization algorithm uses RTT (Round-Trip Time) weighted on each path to reduce latency at single hop level. Experimental results evaluates CAN performances sorting the nodes with respect to the distance from landmarks (topology aware network).

Tests shown that allowing the overloading of coordinate zones improves the performances of CAN, because the impact of node failures is limited. There is a reduction in path length, path latency and per-hop latency because each node has multiple choices for the next node to route the request. Path latency grows very slowly respect to the number of nodes.

Using an improved join algorithm, nodes in the system are uniformly partitioned: more than 80% of the nodes in the system have volume V, that is the ratio between the volume of the d-dimensional space and the number of nodes. The improved join algorithm locates first the node with the largest volume in the system and then performs a join operation on that node, splitting it.

There are no best and worst case experimental values, but for systems with about 2.000 nodes the perceived user query latency is about 2 seconds; this value is lower than 1 second for 256 nodes.

Other papers [SMK+01] point out that the lookup costs increase faster than (log N), that the optimal value of `d` (that is `d` = log N) is good for an unique value of N and that the remapping of nodes into the space after a node failure costs more than claimed by [RFH+01].

Looking at Teleimmersion metrics this low-level DHT-based solution may be very well suitable on Teleimmersion.

4.3. Application level solutions

This section describes the application level solutions: RandPeer, Prefix Hash Tree, range queries over DHT-based systems, support for similarity searches in P2P systems and Skip Tree Graph.

4.3.1. RandPeer

RandPeer manages membership information on behalf of P2P applications, and allows peers to locate good neighbors based on their QoS capabilities, so that applications can construct an overlay network able to meet a certain QoS level [LN05]. It uses a trie data structure (a tree with nodes labeled in a binary way, with label length proportional to node depth) for membership organization and each node of the trie stores the registration entries. RandPeer can split or merge nodes if the number of registration entries becomes respectively too big or too small. The protocols to perform those operations are simple.

The RandPeer data structure, that is a DHT connected like a ring, maps each node of the trie to the RandPeer data structure. RandPeer performances were tested on PlanetLab using CHORD as DHT; for this reason RandPeer is considered an high-level data structure.

Node registration and lookup take O(log log N) steps; latency is higher for limited number of nodes.

4.3.2. Prefix Hash Tree

Prefix Hash Tree (PHT) [RRH+04] is a distributed data structure that implements the support for range queries over a DHT. This means that PHT queries can return all the objects within a certain range, while all the DHT-based querying applications available at the time of the publication supported only exact match queries. PHT uses a trie-like data structure in which each trie node (with label `l`) is assigned to the `HASH(l)` peer; this design choice implies that a PHT node can be

located with one DHT lookup. the binary string of a node PHT also supports Heap queries and proximity queries.

Lookup requires (log D) DHT lookups, where D is the length of the binary string used to index all the elements of the domain, and the insertion and deletion of nodes require one PHT lookup.

Experimental results showed that there is a data loss if a node crashes, therefore PHT is not suitable for Teleimmersion needs, because data consistency is an important metric in the T-MON design.

4.3.3. Range queries over DHT-based systems

The study [GS03] implements range queries in an efficient way on top of DHT-based systems. The solution presents a system to perform range query over parameters of the nodes (e.g., *all the locations with temperature between 20 and 30 degrees*), a functionality offered by MON system (see section 5.5) with non-aggregate queries (the query in the above mentioned example in MON would look like the following: `select locations where temperature > 20 and temperature < 30`).

The data structure is mapped into a tree in which each node corresponds to a load balancing matrix (LBM). There is a load balancing management to keep all the load balancing matrix elements with a similar size. The number of registration messages depends linearly by the number of replicas in LBM.

The research project optimized then the system with regards to some situations: first of all, it provides a path instantiation, that guarantees message passing from parent to children nodes at most once; moreover it has a content pull semantic, that limits the number of queries generated and avoids the broadcast of the tree; the third optimization is the query decomposition, that splits the query in O(log Rq) sub queries (Rq is the query range), making the registration process optimal.

We will not consider this data structure for implementation on Teleimmersion system, because of the lack of any timing measurement for all the metrics.

4.3.4. Support for similarity searches in P2P systems

The study presented in [GS07] creates a support based on DHT for similarity queries on multidimensional datasets; that means that multiple P2P applications can simultaneously use the same DHT, with different dimensions and data distributions. To achieve this result, the system maps a logical kd-tree into a DHT identifier space to form a distributed indexing structure. This is a well engineered solution and allows to reduce a query problem to a distributed tree search problem. The query time is lower than O(log N), that is the value of centralized K-tree.

Evaluation test performs 5.000 random queries over a simulated overlay network of 20.000 nodes and a dataset of 10.000 data points with a fixed number of dimensions. The only notable result is that the number of query messages and the number of hops required to have an high coverage (95% or 100%) decrease exponentially with the number of dimensions of the data set.

Tests did not specify costs for node registration, deletion and distributed tree maintenance.

As mentioned above in section 4.3.3 with regard to Range queries over DHT-based systems, we will also not consider the current solution for implementation on Teleimmersion system, because of the lack of any timing measurement for all the metrics.

4.3.5. Skip Tree Graph

Skip Tree Graph [GSM07] is a distributed data structure that extends skip graphs improving the performances in both exact match and range queries. To achieve this result, the data structure maintains doubly-linked lists at each level, resulting in a set of overlapping skip lists. Skip Tree Graph is based on balanced trees using local operations, that allows a high level of concurrency regarding the insertion and deletion of nodes.

Skip Tree Graph implements the following algorithms: exact-match search operation with conjugates, tree-based exact match search operation, insert operation

and repair mechanism. The research paper reports only the cost for tree-based exact match search operation, that is expected to be O(log n).

The experimental results are expressed as message and hops cost, making hard to evaluate them. The search cost (messages and hops) is about 5 for 250 nodes and 7 for 500 nodes; the insert cost (messages and hops) is about 35 for 150 nodes and about 42 for 500 nodes.

As mentioned above in section 4.3.3 with regard to Range queries over DHT-based systems, we will also not consider the current data structure for implementation on Teleimmersion system, because of the lack of any timing measurement for all the metrics.

4.4. Other DHT-based structures

This section describes the studied DHT-based structures that are nor low-level neither high-level: building layered DHT applications and LiPS.

4.4.1. Building Layered DHT applications

The [CLR+05] scientific paper describes a work that is a case study to build layered DHT applications. The goal of this paper is the porting of the DHT used by an application from FreeDHT to Prefix Hash Tree (PHT). The goal of this work is to study if it is possible to decouple the application level from the implementation of DHT used by the application itself.

At the time of the publication of the paper, the distributed applications required a tight coupling with the DHT they used. The authors try to explore the feasibility of a decoupling between application and DHT layer, even if the results are too much implementation dependent to be used for a generalized scenario.

The DHT level cannot manage the recovery from failure, and therefore it is the application that must address it. DHT is not safe from concurrency failure, so the system relies on recovery management in case of a concurrency failure happens. This

choice increases the flexibility of the proposed solution because applications are free to implement their own policy without being constrained to a low-level policy that may not fit their needs.

One interesting feature is that the lookup does not overload root and high level tree nodes, because it starts at D/2 level. Lookup is doubly logarithmic with the size of the data indexed. There is no information about insertion and delete of elements other than that the PHT grows and shrinks when the system adds or removes keys, respectively.

Experimental results shows that Prefix Hash Tree is balanced, with a maximum tree depth of 33; 80% of nodes have a depth of 26 or lower, and 20% of nodes have a depth smaller than 19. Another relevant result is that block utilization (i.e., the number of elements per PHT leaf node as percentage of the block size) is not linearly proportional to block size (i.e., the storage available on each node).

According to the good software engineering practice to decouple different functional layer, the porting to Teleimmersion of Prefix Hash Tree on Layered DHT Applications as high-level DHT implementation might be a worthwhile solution, because the achieved experimental results are good (even if partially incomplete).

4.4.2. LiPS

LiPS [ZSY+07] is a P2P search scheme based on novel link prediction techniques. The paper lists the properties of social networks: they have a small graph diameter and high clustering coefficient. Those two properties permit to perform efficient searches.

LiPS bases its link prediction on the observation that are existing friends that typically permit to know and meet new friends. The link prediction is static.

The search uses flooding and relies on link predictor accuracy, two factors that make this solution unsuitable for implementation on Teleimmersion system, because flooding is very inefficient with respect to the bandwidth overhead, while Teleimmersion is bandwidth constrained.

Experimental tests shows that the prediction accuracy on best case scenario is just above 60%, a better result than Shortcut [SMZ03], another link-based prediction technique that is very sensible to changes of people's interests.

5. The MON System

MON (Management Overlay Networks) [MON05, JL07] is a lightweight monitor for parameters of nodes targeted for large-scale distributed systems. Its goal is to achieve high coverage, high reliability and low response time using as little system overhead as possible. The typical usage scenario is MON running on nodes where a certain distributed application instance is running so that a MON query can make management of such application easy, knowing which nodes are behaving normally and which ones are deviating, by showing an unusual amount of memory used, CPU utilization or errors in log files. The monitoring can either occur directly if the application implements a defined callback procedure or indirectly through the analysis of system parameters like the values of CPU and memory usage or the content of log files.

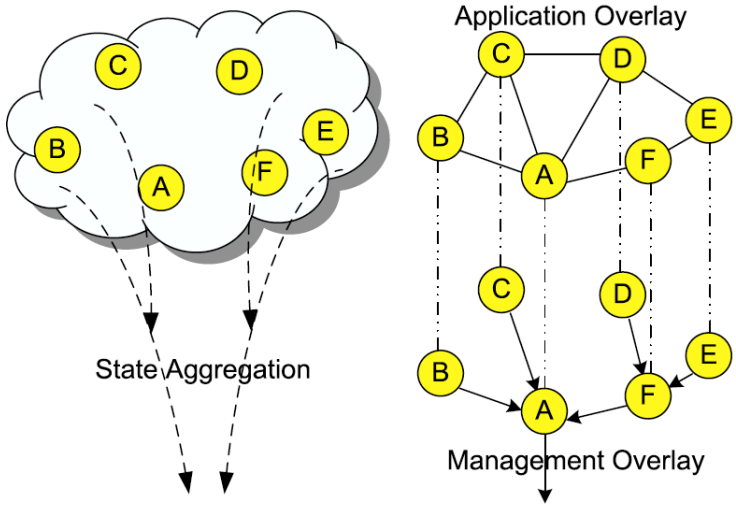


Figure 5.1: Management Overlay network

The left part of Figure 5.1 shows an important capability of MON: it returns aggregate values of queries such as top-K and average, possibly a preferable alternative to the raw data of thousand nodes. The right part of Figure 5.1 shows the on-demand overlay approach: MON hasn't a persistent overlay because MON creates an overlay every time that it receives a query or control command and it deletes it after a short time span.

MON allows monitoring of the system parameters of a node (e.g., CPU and memory usage, free memory, free disk space) and, with the implementation of an API interface, the parameters of an application running on that node. MON presents to the user an SQL-like query language for the node parameters querying. The query language can return the results in an aggregate (i.e., average, top-K) and non-aggregate values, depending on the syntax used to perform the query.

The MON design follows OCMA (Overlay Construction and Maintenance Architecture) architecture for the logical layering structure and PPF (Protocol Plug-in Framework) for the implementation; both OCMA and PPF are targeted for large-scale distributed applications [JL07].

5.1. OCMA Layered Architecture

This section describes the OCMA (Overlay Construction and Maintenance Architecture) architecture, whose goal is to define a reference layered architecture for the design of distributed applications.

Albeit there has been lately a strong research on distributed and P2P systems and applications, little of that work has focused on the leading principles of the design of a distributed application. A distributed application is part of a complex system, so the architecture design is fundamental for the efficiency. As with other complex systems, software engineering proved that layered architecture is the best in terms of system design and innovation, because as long as the layer-to-layer

interfaces do not change, the internal composition of a layer can be changed without impacts on the other layers.

A common design like OCMA (Overlay Construction and Maintenance Architecture) architecture helps defining the structure of the application and the design and role of the different subsystems required for a distributed application. Each layer provides a logical abstraction of a section of the application and performs a specific logical function of the system. Three layers compose the OCMA architecture (see Figure 5.2): Application Specific Processing, Overlay Construction & Maintenance and Membership management.

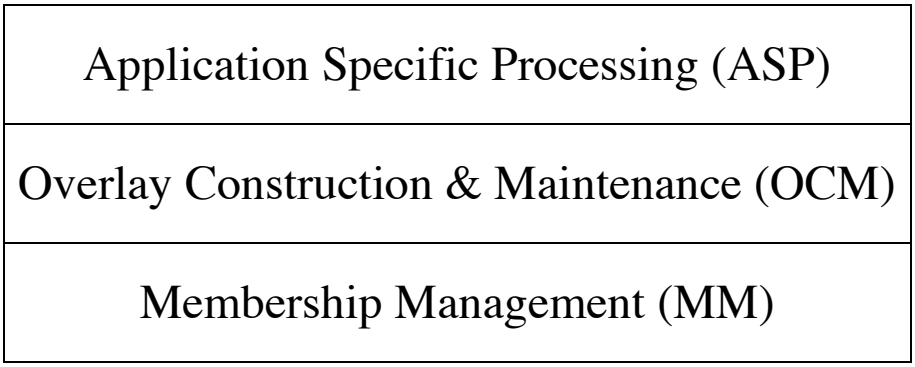


Figure 5.2. The OCMA layered architecture

The ASP layer is the OCMA top layer and represents the data communication, management and processing specific to the application's purpose. For example, for applications that returns distributed queries, the top layer is responsible for propagating the query on the other nodes of the overlay and aggregating the result.

The OCM layer is responsible for constructing and maintaining the overlay structure, which is the logical organization of the nodes of the distributed system within the system itself. The number of nodes in a distributed system changes with time because of failures, so the overlay layer must handle node join and departures. To keep the overlay layer simple, in case of simultaneous multiple node failures the overlay management algorithm might recreate the overlay from scratch (by invalidating the old overlay) instead of locating alternative neighbors: this process might become too complex in case the number of simultaneous failures is high.

The Membership Management layer manages the membership of nodes in the system, with the aim of maintaining high-quality membership information. In a large scale distributed system is difficult to maintain updated information of all the nodes of the system, so a node *p* usually maintains information about a subset of the nodes. This subset of nodes is called *membership view* of *p*, as the nodes in the list are the nodes *p* can communicate with. In an overlay network, a node often needs to communicate only with its neighbors, so to keep a neighbor set that it is usually a subset of the membership view. When a neighbor node fails, the system must quickly locate a new neighbor: the new neighbor selection often selects as new neighbor one of the nodes in the membership view.

OCMA architecture is very coarse grained to leave the developer more flexibility over the design choices of each layer; for the same reason the interfaces between layers are not defined. The developer can also choose to use a solution already available as layer (e.g., for the overlay and membership layers), so that he can focus on the application layer.

5.2. Description of Protocol Plug-in Framework

PPF is a reusable C++ framework for implementing large distributed event driven, single threaded applications. Applications that use PPF can run in simulation (local) and real world (distributed) modes with no extra effort from the developer. PPF requires the use of a single thread because multi-threading introduces non-determinism in the simulation mode; the elimination of non-determinism is possible but the design decision did not consider to implement it to keep the framework simple. The support to multi-threading requires the introduction of mechanisms to guarantee atomicity with respect to the use of shared resources (such as memory and files) and to the code execution (e.g., the event management section). These mechanisms add a significant complexity to the architecture, therefore PPF chooses a simpler approach.

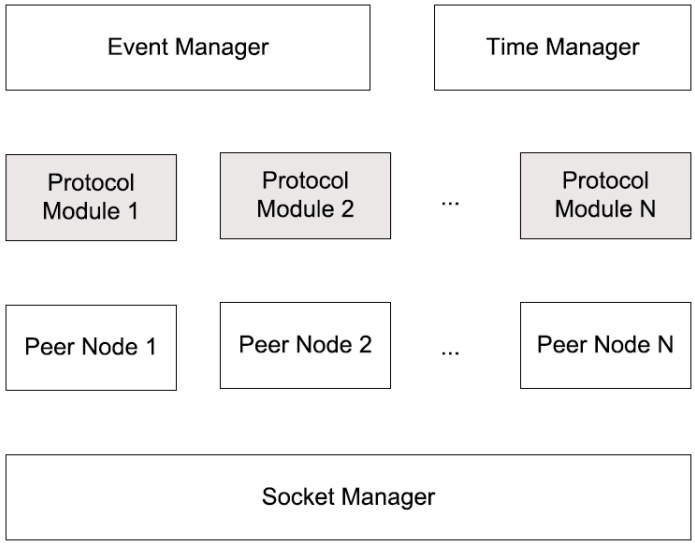


Figure 5.3: Components of a PPF framework

The PPF Framework, shown in Figure 5.3, consists of an event manager, a time manager, a socket manager, and peer nodes, each with one or more protocol modules.

The event manager is the key component and consists of an event queue and a dispatcher. The event queue manages events according to their creation time, so all the events are dispatched in the order of their firing time. The time manager provides time simulation; depending on the execution mode, the time manager can return either a virtual time or the system time.

The protocol module is the main component and the only part that the developer has to implement, by extending an abstract interface provided by PPF. A protocol module mainly provides handlers for timer and network events, like for example the firing of a timer event can cause the sending of a gossip message to some random node, or for a network event the arrival of a message. The design of a protocol module may comply to OCMA architecture, the way it is implemented on MON.

The peer node is the component that provides the network communication interface that protocol modules use to send messages. In simulation mode, messages are scheduled as events, while, in real world mode, the messages are passed to the socket manager that sends messages in the network. All network communications in PPF are asynchronous.

PPF supports separate simulation of time and network, therefore four different simulation combinations are possible. Of these four, the one where the time is simulated and the network is real is not useful because the dispatcher has to deal with two different time representations: the real one for network events and the simulated time for time events.

When the time is simulated the progress of internal time representation is different than the progress of system time, so if the next event is dispatched to fire in the future, the internal time can be moved forward to the next event, making simulation faster. The network simulation is useful to simulate large network applications with little system resource usage (e.g., sockets) and to easily simulate network failures. PPF provides a set of API (for both UDP and TCP communication) that wraps the socket and provides a consistent programming environment for both simulation and real world.

5.3. MON Membership protocol

The membership protocol maintains up to date information about members of the node to avoid the inclusion of failed nodes in the construction of the overlay tree in case of a MON query. This section describes the MON membership protocol, analyzing first the low-level membership management, and then, in the second part, discussing and evaluating three membership view selection algorithms.

As shown in Figure 4.4, MON is deployed in the same environment of the application and is therefore affected by the same node and network failures of the monitored application.

The goal of membership layer is to keep updated information about other MON instances in the system so that the middle layer can use the information to build the on-demand overlay. Each node keeps information regarding *m* other nodes of the system (the membership view, defined in the OCMA architecture); each membership entry contains a logic ID of the node (`peer_id`, unique in the system), the freshness

of the entry (`birth_time`), the IP address (`peer_addr`) and port number (`peer_port`). The age of the entry is the difference between the current time and `birth_time`.

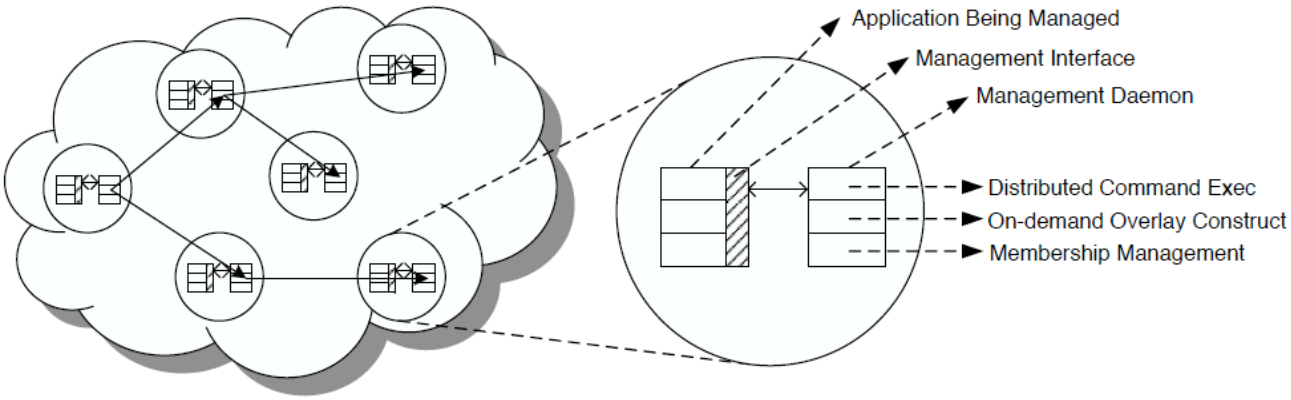


Figure 5.4: MON environment and interface with application

The MON membership protocol is based on gossip [JM+03], so each node periodically sends a `ping` message to a randomly selected node in the membership view. Each `ping` message contains `num_entries` entries from the local membership view (`num_entries` is less than `m`). When the target node receives the message, it merges the membership view received in the `ping` message with its local membership view and then sends back a `pong` message that includes `num_entries` from its membership view. Once the node receives the `pong` message, merges the entries in the message with its membership view.

It's important to note that all the entries sent with a message contains the age of the node and not the `birth_time`, so that the value of the age is independent by the value of local time of each node. Once the node receives a message, it converts the age to the corresponding `birth_time`.

5.3.1. Merge of the membership view with new membership entries

This section describes the merging algorithm between the node membership view and the membership entries received in the membership message.

Once a node either receives a `ping` or `pong` message, it has to merge its membership view with the membership entries included in the message.

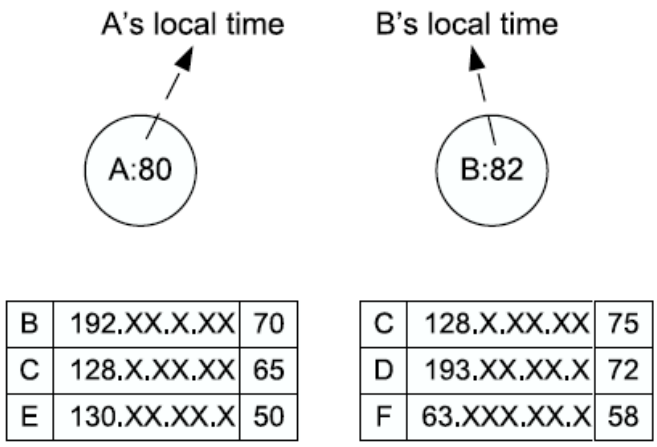


Figure 5.5: Effect of `ping` and `pong` messages over `birth_time` and membership view: before A sends a `ping` (with C and E) to B

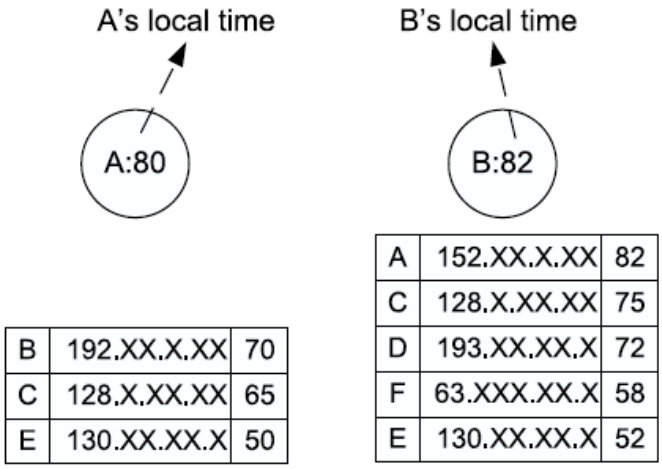


Figure 5.6: Effect of `ping` and `pong` messages over `birth_time` and membership view: after B receives `ping` message

The first step is to check if a received membership entry refers to a node already in the membership view, i.e., a comparison of the `node_id` values. If the node is already in the membership view and the associated `birth_time` value is lower than the `birth_time` of the received entry, the entry in the membership view is updated with the new `birth_time` value (an higher value means fresher entry). According to the example in Figure 5.5 and Figure 5.6, the entry in node A's membership view corresponding to node C is older than the entry of C in the

membership view of B, hence when B receives the `ping` message from A doesn't update the entry referring to node C.

If the received membership entry is about a node that is not on the local membership view, the entry gets added to the membership view. Given that the membership view has an upper limit of entries, if the number of entries in the local membership view is already at the maximum value allowed, the algorithm randomly deletes an entry from the local membership view and replaces it with the new entry from the received membership view. If the age of a node is higher than a defined threshold, the corresponding entry is deleted in the membership view.

Experimental results confirmed that the higher is the number of membership entries sent with a message, the lower is the average age of the entries.

5.3.2. Membership view selection

This section describes the characteristics of the greedy and hybrid view selection algorithms, then compares them to the random view selection algorithm and, according to experimental results, selects the algorithm that MON implements.

The algorithm for the selection of the entries to send in the gossip message influences the average age of the entries and therefore affects the performances of the system. The MON research [JL07] proposed and evaluated two new membership selection algorithms: the first algorithm, Greedy, selects the entries with the smallest age. The second algorithm, Hybrid, splits the membership view in two parts: half with the fresher entries (fresher half) and the older half. The algorithm selects the entries included in the gossip message between the entries in the fresher half.

The MON research implemented and compared those two algorithms to a pure random membership entry selection. The test system has $N = 1000$ nodes, each node keeps $m = 40$ entries in its membership view and gossips every $T = 10$ seconds. The average age for each algorithm is shown in Figure 5.7.

The average age of Greedy and Hybrid algorithms decrease faster than the pure random because the membership information propagated is more fresh.

To determine which algorithm provides the most uniform sampling of the system, the project has simulated a system with fixed neighbors (with the same parameters as the previous simulation). The in-degree distribution of the nodes was selected as metric, providing the results shown in Figure 5.8.

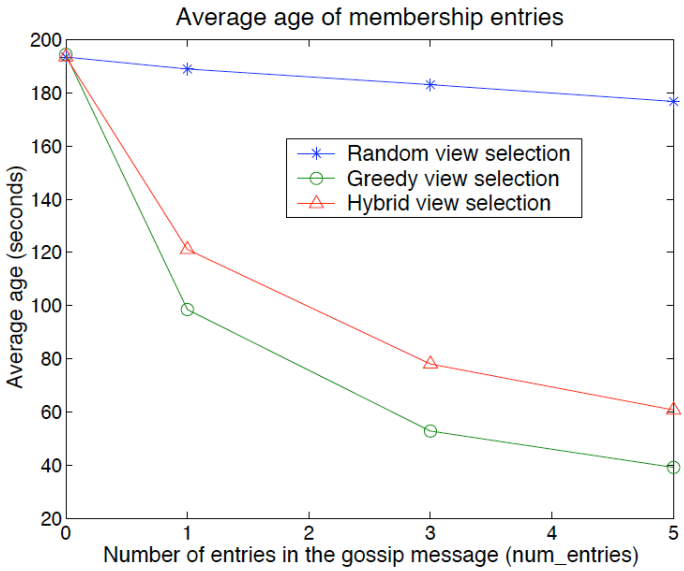
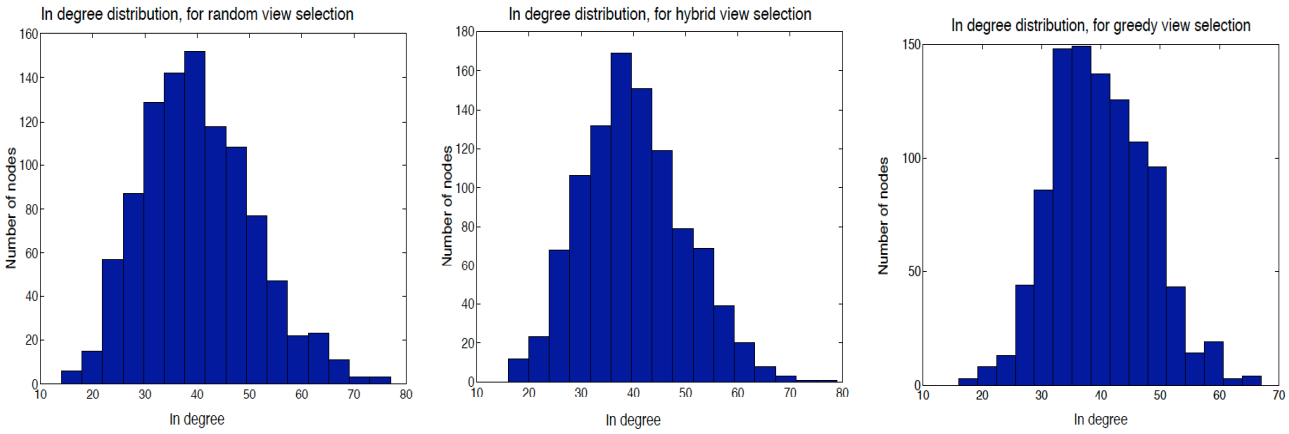


Figure 5.7: Average age of membership entries



(a) Random view selection (b) Hybrid view selection (c) Greedy view selection

Figure 5.8: In-degree distribution of the three membership view selection algorithms

The Random selection is considered the reference value among the three algorithms because it is likely that the membership view of any node represents a uniform sampling of the whole system. The greedy and view selection algorithms

may affect that property, even if they decrease the average membership age. The Hybrid view has a greater in-degree distribution similarity to Random than Greedy, i.e., the randomness of the membership view in the Hybrid view is higher than in the Greedy view. Therefore the selected membership selection algorithm for MON is Hybrid, despite Greedy provides entries with a smaller average age.

5.4. Overlay construction

This section describes how MON builds the overlay network on demand. Given that an overlay tree is ideal for distributed status query and control, the analysis will focus on tree-based overlays. To provide redundancy, MON creates on-demand DAGs (Direct Acyclic Graphs).

The node that receives the query or the service command becomes the root of the tree. Each node selects `k` nodes (called children nodes) and sends a propagate tree message (`session` message) to each of them. When a node receives a `session` message selects `k` children nodes and repeats the procedure. When a node receives a `session` message for the first time it replies with a `session_ok` message, otherwise it replies with a `prune` message. When a node receives `session_ok` from all its children, it sends the `session_ok` message to its parent node. Once the user receives a `session_ok` message, the overlay tree is created.

It is possible to use different algorithms for the selection of children nodes, providing different performances. Two metrics evaluate the performances: the percentage of live nodes included in the overlay tree (coverage) and the time between the command is sent and the time a result is returned (response time). There is a direct correlation between the response time and the time used for the creation of the overlay.

We describe and evaluate three specific node selection algorithms: `randk`, `leafset+RF` and `tree+RF`.

5.4.1. The `randk` Algorithm

The `randk` algorithm propagates the session message to random nodes selected from the membership view. The algorithm is very simple (based on the membership based gossip protocol) and provides a probabilistic coverage. Therefore the only way to achieve high coverage is to use high values of `k`.

5.4.2. The `leafset+RF` Algorithm

The `leafset+RF` (Random Forwarding) algorithm adds deterministic forwarding for the tree construction message: each node, in addition to the membership view, maintains a list of `l` nodes whose `node_id` are closer to the id of the node (leafset). Half of the leafset contains nodes with higher `node_id`, while the other half contains node with a lower value. In the example showed in Figure 5.9 node H has failed but is still in the membership view of nodes I and G. The one-directional arrows represents the link to H (failed node) in the membership view of the nodes G and I

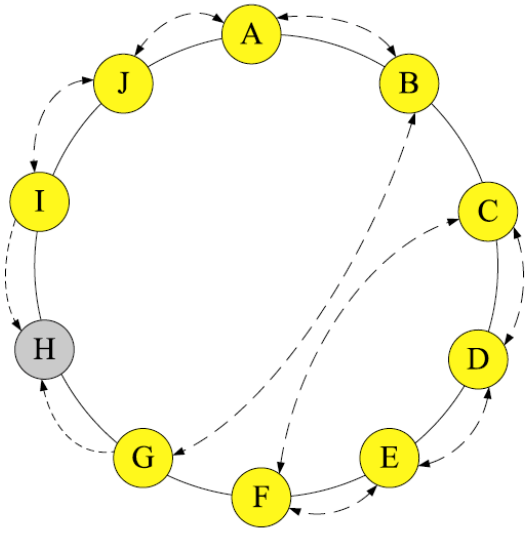


Figure 5.9: Example of leafset

The creation of the children list of the node occur by selecting `k'` nodes from its leafset and `k – k'` nodes from its random membership view. The name corresponding to the message propagation is leafset forwarding and random

forwarding, respectively. A high value of `k'` increases the coverage but reduces the depth of the overlay tree, and considering that the value of `k` is constant, increasing `k'` reduces the random forwarding degree. The algorithm provides complete coverage with high probability.

This algorithm performs better when is possible to assign similar node IDs to nodes that are close respect to the network (e.g., nodes in the same datacenter), but this node ID allocation method decreases the strength of the system to node failures.

`leafset+RF+LOC` (LOCality awareness) algorithm is a variant of the `leafset+RF` algorithm where the network delay between nodes is the metric used to list the selection of the children, under the (simple) assumption that a smaller network delay is directly associated with a geographically closer node.

5.4.3. The `tree+RF` Algorithm

The `tree+RF` (Random Forwarding) algorithm is based on the `leafset+RF` algorithm where the membership layer maintains a tree structure in addition to the random membership view, hence each node contains a parent and many children neighbors. Persistent trees are hard to maintain, but the tree structure described does not require to be continuously accurate: it can have loops or be temporarily partitioned. This algorithm creates the overlay using the same procedure described at the beginning of the section, sending the `session` message to `k` children nodes.

5.4.4. On-demand DAG construction

In case of node failures the overlay tree becomes partitioned, that is undesirable. To overcome this risk, is useful to build a DAG overlay. The tree overlay algorithm can create the DAG overlay just with the addition of a step: each node propagates the construction messages as before, accepting more than one node as parent. Whenever a node receives a construction message and has enough parents replies with a `prune` message. The resulting overlay structure is a DAG, making less likely that a live node may be disconnected.

5.5. MON query language

This section describes the MON query language, a relevant topic for the purpose of the T-MON project, that mainly focuses on the extension of MON query language, as illustrated in section 7.2. Therefore the following analysis is very detailed and explains the three typologies of MON query: aggregate, non-aggregate and other queries. MON query language also allows the execution of some system commands (e.g., grep) on all the MON nodes.

The MON query propagates through the system in the following way: once the creation of an on-demand tree is completed, the query or control command is propagated through it and executes. By starting from the root node (i.e., the MON node that receives the query from the user) the command propagates to every child node. Each node executes the command, and once the results are received from all its children it packs all results and send the aggregate data to its parent node.

To create a DAG overlay each node has to select the primary parent, that is the only node it sends the aggregate data to, so that every result has only one return path.

The MON query language presents a SQL-like syntax, providing a database-like view of the distributed application to the developer.

The following sections present some different typologies of MON queries.

5.5.1. Aggregate queries

The MON aggregate queries can compute aggregate functions (such as average and top-K) on the system. The general language syntax for the aggregate queries is the following:

```
select agg(<resource>) [where <bool_expr>]
```

Where `agg` is one of the aggregate functions supported: `avg`, `top-k` and `histogram`. The `resource` is the metric the user wants to query, like for example `filesize("mon.log")` queries the size of the file with name mon.log, and

`procmem("mon")` and `proccpu("mon")` query the memory and CPU usage of the process mon. Other possible aggregate queries are the following:

```
select count [where <bool_expr>]

select avg(<resrc>) [where <bool_expr>]

select top <num> <resrc> [where <bool_expr>]

select bot <num> <resrc> [where <bool_expr>]

select histo2 <minv> <maxv> <num_bins> [where
<bool_expr>]
```

The `where` section is optional and, if it is included in the query, the `bool_expr` (also defined as `condition`) is evaluated first; only if the command is true the command is locally executed. The condition is expressed in CNF (Conjunctive Normal Form), as `and` of different conditions.

5.5.2. Non-aggregate queries

The non-aggregate queries have the following structure:

```
select <resrc1> [, ...,  <resrc_k>] where <bool_expr>
```

The resources (`resrc`) available for querying are the following: `busycpu`, `freemem`, `diskusage`, `freedisk`, `uptime` and `load`. The Boolean expression is a combination of one or more conditions in the CNF form. The conditions have the form of `<resource> <op> <value>`, where the `op` is a standard operator (i.e., greater than, equal to, less than, etc) and `value` is a natural non-negative number. An EBNF (Extended Backus–Naur Form) representation of the grammar for the `bool_expr` is illustrated in Figure 5.10.

```
<boolexpr>    ::= <shortexpr> | <longexpr>
<shortexpr>   :: = <param> <op> <val> | "(" <shortexpr> ")"
<longexpr>    ::= <shortexpr> and <shortexpr> { and <shortexpr> }
<op>          ::= >{<eq>} | <{<eq>} | <eq>
<eq>          ::= =
<val>         ::= <digit> | <nzdigit> { <digit> }
<digit>       ::= 0 | <nzdigit>
<nzdigit>     ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 5.10: EBNF grammar of `bool_expr`

Non-aggregate queries have the `where` clause mandatory to limit the number of results returned by a query.

5.5.3. System commands and other queries

MON also supports status control commands that are able to run any shell command on all the nodes, a situation that may be exploited when performing distributed deploy: once a specific update script is sent to each node, automatic effortless updates of all the instances are possible.

The general syntax is the following:

```
select run(<cmd>) [where <condition>]
```

The query executes the command `cmd` on all nodes matching the `condition`, if any.

Some high level commands such as grep have their own implementation into MON as illustrated in the following query:

```
select grep(<keyword>, <file>) [where <condition>]
```

The distributed grep of application log files is also referred to as distributed log query and is a way to indirectly grab application status information.

MON also supports the capability of doing basic deployment.

6. Toward a T-MON implementation

This chapter will explain the set-up of T-MON environment, mainly focusing on PlanetLab and related management scripting, and the analysis of MON code.

After having studied all distributed systems listed in Chapter 5 and a careful comparison of MON features, we have decided to use MON as base platform for the implementation of a distributed monitoring, control and deploy platform for TEEVE system. There are many reasons behind this choice. Starting from scratch would have meant adding uncertainty to the project because the development of a distributed support infrastructure, like any software project, may incur in several issues that may lead to delays: software bugs require time to be fixed and sometimes they require to redesign an entire section (or logical section) of the software. This particular project was composed by a single person, i.e., no other people working on similar projects, so there is no team to eventually compensate problems. The final distributed system might also have different performances than expected, requiring more resources like bandwidth or CPU time (two resources that are preciously needed by TEEVE system). The selected solution must have as little overhead as possible, particularly with respect to those two metrics. The technical papers describing the distributed systems considered in Chapter 5 often highlight only the positive aspects of the system described and rarely show its drawbacks; sometimes the drawbacks are highlighted in papers describing a newer system. This uncertainty would have required a live comparison between the few systems that might have been suitable for the TEEVE needs to completely evaluate and compare them (including understanding the weaknesses of each system), and this is a procedure that requires time.

On the other hand, MON is a mature project: it has been developed, tested, is stable and has a defined overhead and resource requirements, that provides a complete infrastructure from the overlay and membership management to the application level. This allows focusing on the core goal of the project rather than

spending time for the development of the distributed communication layer. Reusing components is also a good software engineering practice.

For the above mentioned reasons we have defined MON as base application, hence the project was named T-MON (Teleimmersion – MON).

The first actions were about analyzing MON code and its environment (PlanetLab), setting a personal programming environment consisting of a set of scripts to deploy, running and managing MON on PlanetLab, a SVN repository to manage the code and a test server where test code changes before live deploy. In parallel to that, a deeper study of Teleimmersion system and talks with students of the TEEVE project's group to ask them which TEEVE parameters they would like to be able to monitor. This activity regarding the study of TEEVE and the T-MON query language parameters will be discussed in Chapter 7.

6.1. PlanetLab environment

PlanetLab [PL02] is a distributed network for testing distributed applications and system in a wide scale real-world environment. It is purely used for research purposes and many Institutions from all the continents contribute to the network providing one or more servers. Currently there are about 900 nodes from more than 450 locations worldwide, as illustrated by Figure 6.1.

Each location provides either one or several nodes to the network and in return the Institution has the capability to use all PlanetLab nodes for its research needs. Each account is called a slice, a sort of virtual machine, that is isolated from the other slices running on the node. Each slice has a portion of the resources of the nodes: CPU usage, available RAM, disk space, inbound and outgoing bandwidth, number of connections. An average application doesn't exceed the PlanetLab's Acceptable Use Policy (AUP), but in case the research application needs extra resources, it is possible to increase the limits (e.g., if the application opens many simultaneous connections to IRC servers, PlanetLab suspends the slice until the owner provides an explanation).

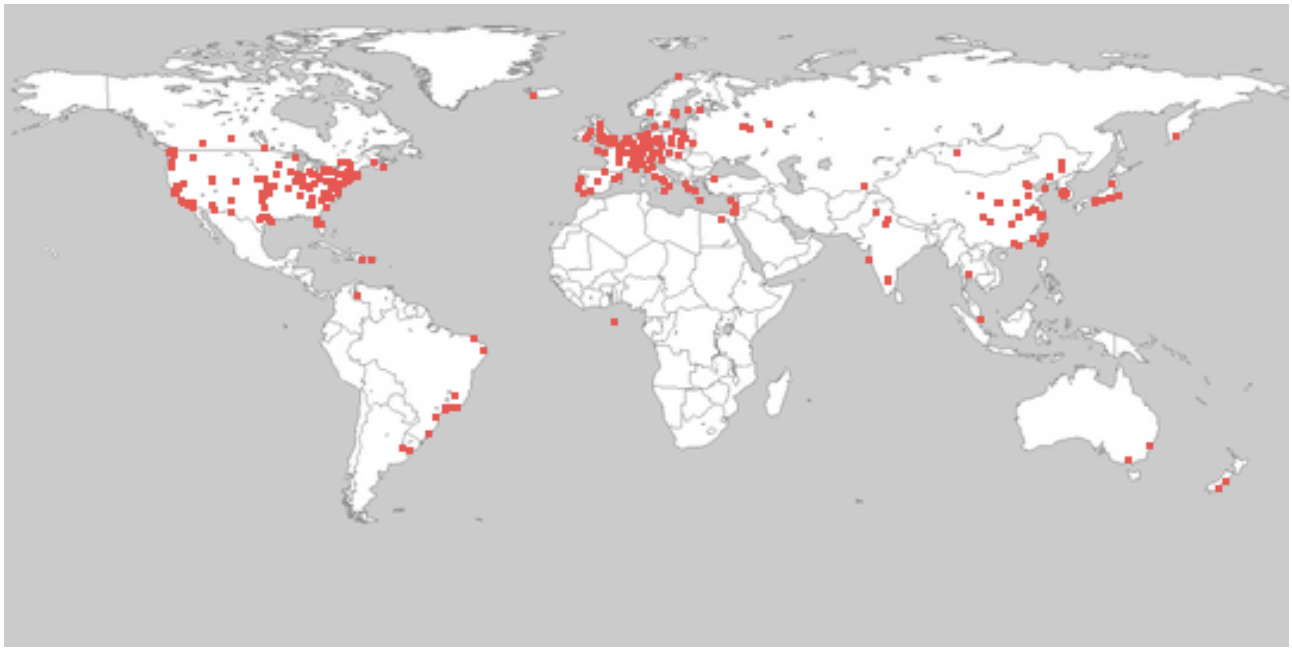


Figure 6.1: Map of PlanetLab nodes location

PlanetLab nodes run a specially modified Linux distribution, called PlanetLab OS, that includes the management of users, slices and Institutions, the management of the permissions, logging and abnormal activity monitor. PlanetLab OS is based on Red Hat 4.1, released on June 2005. The current development is moving the next major release of PlanetLab OS to CentOS 5.2 [PLDEV].

After the site's PI (Prof.Klara Nahrstedt and Prof.Indranil Gupta for University of Illinois account) have created the slice and the user account, the user has to log into the personal area of PlanetLab to activate the slice and wait up to 48 hours to get it spread across the PlanetLab nodes. Any slice is by default associated to about 20 nodes, probably randomly selected to balance the overall load over the system. This choice also quietly tells users to use only nodes if they really need for them; a confirm comes from the slice registration page, where there is no option to easily associate all the nodes to the slice, but the user has to click to one checkbox for each node. For a project like MON is important to have as much nodes as possible, hence all the nodes have been associated to the slice. The list of PlanetLab nodes changes frequently and new PlanetLab nodes are not associated to the slice by default, so the

user must periodically check the page. To avoid the expiration, the slice must also be refreshed, with a renewal interval of up to two months.

To log into the slice, the user has to generate a RSA key pair and upload the public key to PlanetLab from the "key management" section of the user's private area.

Once keys are propagated in the PlanetLab system (it may take up to 48 hours) the user can log into the PlanetLab nodes using the ssh [SSH] protocol. The command to connect to the `planetlab2.uc.edu` node on the `uiuc_monquery` slice is the following:

```
ssh -l uiuc_monquery -i ~/.ssh/id_rsa planetlab2.uc.edu
```

Updates to the PlanetLab online documentation are rare, therefore a web search is often the best instrument to find the answers. An alternative is to email to PlanetLab Support, very helpful and with reasonable fast reply time.

The security policy of Computer Science Department at the University of Illinois does not allow to save users password on computers located outside the campus network. The PlanetLab slice on each node is accessible only to the authorized users, but the administrator of the node has full access to the data of all the slices of that node, so all the scripts and deployment planning had to take this into account.

6.1.1. Vxargs

Vxargs [VX04] is a tool written in Python to run the same command in parallel to a list of nodes, provided as either IP addresses, or hostnames, or even mixed. The list is a text file or a system input (generally a pipe) and each line represents a node. It is useful to control a large set of machines over a wide area network, providing a real-time visualization of the execution of commands and a complete report of its execution.

Vxargs can run any shell command; the most used commands in PlanetLab environment are ssh, scp [OSSH02] and rsync [RSYNC]. Vxargs needs at least three arguments to execute: the reference to the list of nodes, the folder where to save the final report and the command to run. The command must have the *{}* sequence in place of the address so that Vxargs dynamically replaces it with the hostname. The sequence can appear one or more times in the command and all the sequences will be replaced by the hostname at run time. The command illustrated in Figure 6.2 checks the uptime of a set of nodes listed on the file iplist.txt:

```
vxargs -a iplist.txt -o /tmp/result ssh -l uiuc_monquery -i
     ~/.ssh/id_rsa {} uptime
```

Figure 6.2: Simple Vxargs execution

6.1.1.1. Vxargs report folder

The report contains three files for each node: `node.out`, `node.err` and `node.status`. `node` is the entry exactly as it appeared in the list provided to Vxargs, so may be an IP address or a hostname (e.g., when Vxargs performs an operation on node `planetlab1.cs.uiuc.edu`, the final report folder contains the files `planetlab1.cs.uiuc.edu.out`, `planetlab1.cs.uiuc.edu.err` and `planetlab1.cs.uiuc.edu.status`). The `.out` file contains the output from the `stdout` of the remote node, the `.err` file contains the output of the remote `stderr` and the `.status` file contains the exit value of the command launched by Vxargs (a positive integer). There is one `abnormal_list` file that contains the list of nodes that failed to execute the command (In the example of Figure 6.2 the file would list all the hosts where the ssh command failed).

6.1.1.2. Automatic key authentication for Vxargs

Every connection to PlanetLab requires a password. The authentication is not password based, but a password is required to use the user's private key. It is technically possible to use a passwordless private key, but PlanetLab discourages it,

because, if the key is stolen, attackers would have access to a very big infrastructure as PlanetLab is. Vxargs doesn't offer the capability for the user to type passwords, hence Vxargs has to run in an environment where the private key has been authenticated once and the password prompt will never appear until the environment runs. See Figure 6.3 for the list of commands.

```
eval `ssh-agent`
ssh-add ~/.ssh/id_rsa
cd ~/monproject_link/mon/vxargs
vxargs -a iplist.txt -o /tmp/result --timeout=600 -P 16 ssh -l
     uiuc_monquery -i ~/.ssh/id_rsa {} uptime
exit
```

Figure 6.3: Use of Vxargs without password prompts

`ssh-agent` is a program to hold private keys used for public key authentication, while `ssh-add` adds the key to the authentication agent (i.e., `ssh-agent`). `ssh-add` asks for the private key password. The `exit` command is required to exit the environment and invalidate the automatic authentication [EXIT].

Figure 6.3 shows the typical Vxargs options used during the project. In particular, the timeout was set to 10 minutes (600 seconds, default value is 300) and the number of concurrent execution was limited to 16 (`-P` option, default value is 30).

If the user forgets the password of the private key or if someone steals the key there is the possibility to recreate a new key pair and upload it to PlanetLab through the user's private area.

6.1.1.3. Vxargs usage modes

There are three different ways to use Vxargs to automate execution: the first one is the one shown in Figure 6.2, where there is one connection per host. When the connection with the host is closed, the needed operations on that host are completed. It is important to note that in case of ssh, multiple commands can be executed within

the same session. For example, in the execution of Figure 6.2 the following string can replace the command `uptime` (note the inclusion between single quotes):

```
'killall mon; ./mon/remote_start_mon2'
```

The launch of the above commands inside a Vxargs-based ssh execution performs a basic refresh of MON instances across PlanetLab node.

The other two typical usage scenarios happen when there is the need to execute two or more operations on all the nodes, but each operation requires two different commands. Imagine that a new executable has been compiled and needs to be deployed to PlanetLab nodes. The use of a single command would be possible if the node would receive the executable from a central server (pull mode) using for example scp or rsync commands. The use of those commands would require to either save the password on the node or write it in the command as argument (i.e., saved in the node logs). Therefore this possibility doesn't comply with the security policy of Computer Science Department at the University of Illinois described in section 6.1. Hence there's the need of at least one file copy command (e.g., scp or rsync) and one ssh to launch the new executable.

The second way to use Vxargs is to create a script where the different Vxargs commands are executed sequentially. This means that first the scp is executed on all the nodes and then Vxargs runs the ssh on all the nodes, as shown on Figure 6.4; for this reason we will refer to it as non-atomic Vxargs. This procedure is sometimes unacceptable, because PlanetLab nodes may fail or join the network between the two Vxargs executions, so there may be nodes where the scp is successful and ssh fails and other where the scp fails while ssh is successful. Those two situations are unacceptable in some cases, so we designed a third Vxargs execution method (referred to as atomic execution). The atomic execution, shown on Figure 6.5, forces Vxargs to wait until the execution of entire script, guaranteeing that all the commands included in the script connects to the same node in a short period of time, minimizing the probability of a node failure during the execution.

```
#!/bin/bash
#usage: ./vxargs_doubleexec.sh
vxargs -a iplist-full.txt --timeout=420 -o /tmp/vxargs/result-scp
     scp -i ~/.ssh/id_rsa -o StrictHostKeyChecking=no -r
     pl_folder/systemFiles/yum.conf uiuc_monquery@{}:
vxargs -a iplist-full.txt --timeout=3600 -o /tmp/vxargs/result-ssh
     ssh -l uiuc_monquery -i ~/.ssh/id_rsa -o
     StrictHostKeyChecking=no {} 'sudo mv yum.conf
     /etc/yum.conf;sudo yum -y install which; sudo yum -y install
     mysql-server'
```

Figure 6.4: Bash script that `vxargs_doubleexec.sh` executes Vxargs in non-atomic mode

The main difference from the user point of view between atomic and non-atomic execution is that in the atomic mode there is a direct invocation of Vxargs, while in the non-atomic mode Vxargs invocation is wrapped inside the script.

```
#!/bin/bash
#usage: vxargs -a iplist-full.txt --timeout=1200 -o
     /tmp/vxargs/result vxargs_doubleexec.sh {}
scp -i ~/.ssh/id_rsa -o StrictHostKeyChecking=no -r
     pl_folder/systemFiles/yum.conf uiuc_monquery@"$1":
ssh -l uiuc_monquery -i ~/.ssh/id_rsa -o StrictHostKeyChecking=no
     "$1" 'sudo mv yum.conf /etc/yum.conf;sudo yum -y install
     which; sudo yum -y install mysql-server'
```

Figure 6.5: Modification of `vxargs_doubleexec.sh` bash script that executes Vxargs in atomic mode

The atomic mode usually requires higher timeouts than the other Vxargs execution modes.

6.1.2. Management of PlanetLab host list

After the overview of PlanetLab and its tools, it should be clear the importance of having and maintaining a good host list, i.e., a list without unreliable or test nodes. For this reason we prepared appropriate node management scripts.

The first step was to prune and exclude the unreliable or unavailable nodes. We launched four simple Vxargs run (similar to the one of Figure 6.2) on different days

of the week and with different start time to avoid maintenance downtimes. We set up a list of the nodes that failed at least three times with the `badhosts.sh` script. The script used as input the Vxargs log folders and looked for hosts who had the value of 1 in the corresponding `.status` file. We compared the resulting list of unreliable nodes to the initial PlanetLab node list to remove the entries corresponding to the unreliable hosts. The script, written in Perl, who removed the unreliable nodes from the list is `removeBadhosts.pl`. Once we have pruned the list of PlanetLab nodes from unreliable nodes, we observed that the hosts belonging to the same Institution were not grouped together, i.e., the file containing the list of nodes had the entries unsorted. To model Teleimmersion efficiently with T-MON the idea was to deploy T-MON instances representing a Teleimmersion site on nodes belonging to the same Institution with the assumption that those nodes would have been connected to each other in a LAN and in a WAN to other PlanetLab nodes, providing an excellent network simulation for the Teleimmersion environment.

A Perl script, `sorthosts.pl`, grouped all the nodes by hostname so that the hostnames with a common domain would show up in the list closer to each other. The sorting algorithm indirectly grouped the host names by their top level domain name (TLD), making easier to select PlanetLab node clusters geographically distant.

We discovered that the MON executable accepts only a list of nodes with IP addresses, and that no hostnames are allowed. A Python script, `pygethostbyname.py`, converted the hostnames-only list into a list with only IP addresses; the list generated became the MON `memberlist` file.

6.1.3. Software repository and installation

The PlanetLab OS uses yum for package management due to the fact that it's a RedHat-based Linux distribution [PDN023]. We tested the functionalities of yum on PlanetLab trying to install `which`, a simple program to locate a command that is in the system PATH.

Yum randomly failed to install `which` on a significant portion of nodes with no apparent reason. Further investigation pointed to the PlanetLab's package repositories: PlanetLab support staff claimed that they were very unreliable and therefore the suggestion provided was to change the configuration to include more repositories. Finding a RedHat 4.1 (an old release) package repository was an hard task and the use of a newer repositories (i.e., repositories for following RedHat Linux releases) proved impossible: yum downloads first from the repository an xml file that includes the RedHat version the repository is for, and if the repository is for a following version, yum skips that repository. Some packages have a PlanetLab OS specific version, so yum refused to install all packages on certain nodes; until the PlanetLab repository didn't came back online, we had to halt the management of that node.

Figure 6.5 shows the Vxargs script used to copy a new `yum.conf` to the node and then overwrites it to the default file and runs the installation of some packages.

Independently from the repository selected, it proved impossible to install Subversion (SVN) on any PlanetLab node. SVN would have simplified the Vxargs script, as it would have required only one ssh connection per node to pull the modified binary executable.

The `uiuc_monquery` slice was mistakenly deleted on January 2009. Albeit the PI promptly recreated it, the repository substitution commands described in Figure 6.4 and Figure 6.5 did not succeed.

6.2. Development server for testing

Vxargs opens many ssh connections in a short time span, a behavior that can be detected as suspicious activity from a compromised computer. We performed the first experiments with Vxargs on a private laptop connected to the University network, and it soon had been disconnected from the network.

We then decided to use a server belonging to MONET Research Group. We created the development account on the server `miami.cs.uiuc.edu`, simply known as miami server. Since miami was a research server managed by the University staff, the amount of connections that Vxargs created was not a problem. The only additional step required was to connect on ssh into miami and execute Vxargs from there.

A development server proved useful to test the executables before deploying them to PlanetLab (the PlanetLab staff strongly discourages the deploy of untested applications).

Miami was running an old version of Linux (RedHat Linux 8 with 2.4 kernel) and apparently nobody in the Research group was the administrator of it, so a grace period of some days had to be set up before the beginning of a complete OS reinstallation that installed RedHat Enterprise Linux with a 2.6 kernel. The machine included the installation of SVN as well as other software.

Miami does not use `yum` for package management, but the new server installation included all the required software; we had to carefully maintain the version of each package to the same version available on PlanetLab (e.g., for `mysql` package).

6.3. Analysis of MON code

The University of Illinois provided a SVN account, that supported MON development, making the development more flexible [TSG07].

MON is a fairly big and complex project written in C++. It consists of six root folders: `common`, `libmon`, `managed`, `membership`, `mgmt` and `session`. The `common` folder contains the code of common tasks needed by MON (among them: packet management, timer, event manager and new array types), while the `libmon` folder contains the files and libraries to interface an application through MON API. The `managed` folder contains the main class and the Makefile, the `membership`

folder contains the code related to the MON membership management layer, while the `mgmt` folder contains the data aggregation functions (such as top-K and average). The session folder include the files to manage a query, including the parsing and query execution.

The analysis of MON code proved to be harder than planned because little or no detailed code documentation was available, the comments in the code were often not complete and the naming of few classes and variables was ambiguous, leading to confusion and consequent analysis mistakes. The only documentation available (a Ph.D. Thesis [JL07] and some research papers describing MON) addressed the code at the high level and did not include the description of topics such as the execution flow, the query parsing and the socket communication. The complete syntax of the query language was disseminated between the cited documents and the source code. The MON developer left the University of Illinois after his graduation and at the time of the T-MON development nobody had knowledge about MON code. The emailing with the MON developer provided some help.

6.3.1. Analysis of MON execution flow

The network and simulation execution mode have the same flow in accordance with the Protocol Plug-in Framework. The following data flow analysis refers to the main execution flow (create a session, submit one or more queries, exit); there are multiple possible execution paths depending on the state of the application and the events in the event queue (e.g., the overlay and membership management) that we will not consider in this analysis.

The main function is in the `managed` folder (for this reason the MON executable has the name `managed`) in the `main.cc` file. The main performs the check of the invocation parameters and the initialization of the environment, then invokes the event monitor loop, `DoLoop`. The `DoLoop` periodically checks if there are incoming messages with the `TryReceiveMessage` function. If there is a message, the `MgmtClient::HandleCommand` function (located in `session/`

mgmtclient.cc) is invoked with the content of the received message buffer as argument. The HandleCommand has a series of if-else if-else if statements (semantically equivalent to the case-switch statement) over the content of the buffer. This is one of the places in the code where the complete message syntax (mainly control messages) is visible. The relevant part for this analysis, the check for the MON query (i.e., the buffer starts with select), is illustrated on Figure 6.6. That Figure shows that the following step is the invocation of the parsing of the query with the parse method.

6.3.1.1. Parsing of the query

The parsing follows a semantic similar to the StringTokenizer of the Java programming language, that outputs the substring between the current position and the following separator. In MON the separator is the whitespace character, therefore all the substrings between whitespaces are token. The MgmtCommand::parse parsing function uses the methods of TokenReader class defined in the session/tokenreader.h file. This file contains a list of definition of tokens as macros with the suffix TK_ (e.g., TK_KEYW_SELECT, whose associated value is 38).

```
[…]
}else if(!strncmp(buf, "select", 6)){
      //all "select" type of commands parsed here
      MgmtCommand mgmt_cmd;
      if(mgmt_cmd.parse(buf)<0) {
         sprintf(str, "Unknown command %s\n", buf);
         result.status = MONRESULT_UNKNOWNCMD;
      }else StartTask(&mgmt_cmd, &result);
   }else […]
```

Figure 6.6: Section of interest in MgmtClient::HandleCommand function

The TokenReader class has a variable named keyw_table that is an array of keywords. A keyword is a pair <string, TK_TOKEN> where the string is the

token of the query string and the TK_TOKEN is the macro to associated to the token. The `keyw_table` is illustrated in Figure 6.7.

```
static keyword keyw_table[] = {
  keyword("and", TK_KEYW_AND),
  keyword("or", TK_KEYW_OR),
  keyword("count", TK_KEYW_COUNT),
  keyword("avg", TK_KEYW_AVG),
  keyword("top", TK_KEYW_TOP),
  keyword("bot", TK_KEYW_BOT),
  keyword("histo2", TK_KEYW_HISTO2),
  keyword("where", TK_KEYW_WHERE),
  keyword("maxmissing", TK_KEYW_MAXMISSING),
  keyword("select", TK_KEYW_SELECT),
  keyword("topology", TK_KEYW_TOPO),
  keyword("depth", TK_KEYW_DEPTH),
  keyword("path", TK_KEYW_PATH),
  keyword("run", TK_KEYW_RUN)
};
```

Figure 6.7: The `keyw_table` array in `session/tokenreader.cc`

The parser checks that the first token is a `select` and then a `case-switch` statement detects the types of all the other tokens. Every selection but the default one invokes a method whose names begins with `MgmtCommand::ReadCmd`.

6.3.1.2. Non-aggregate queries

If the query is a valid non-aggregate query, there is an invocation corresponding to the specific aggregation function (e.g., `ReadCmdCount` for the count function); otherwise the token is `TK_NAME` and therefore the `ReadCmdSel` method is invoked. The `ReadCmdSel` method performs a second parsing of the token through the invocation of the `ResrcSensor::GetResrcType` method, that associates the token to a resource macro. The `mgmt/resrc.h` file contains the definitions of the resource macros, that begin with `RESRC_`. All the allowed keywords of the MON

query language but the ones defined in the `keyw_table` are resources. The `GetResrcType` function provides the mapping between a token and a resource, while the `GetResrcName` function performs the mapping between the resource and its string representation. Sample parts of those methods are illustrated in Figure 6.8.

```
const char*ResrcSensor::GetResrcName(short resrc_type)
{
  char* p = "Unknown resource";
  switch(resrc_type){
     case RESRC_NULL:
        p = "NULL"; break;
     case RESRC_VERSION:
        p = "VERSION"; break;
     […]
     default:
        ;
   }
   return p;
}

int ResrcSensor::GetResrcType(const char*name)
{
   if(!strncmp(name, "load5", 5)) return RESRC_LOAD5;
   else if(!strncmp(name, "load", 4)) return RESRC_LOAD1;
   else if(!strncmp(name, "version", 7)) return RESRC_VERSION;
   […]
   else if(!strncmp(name, "delayto", 7)) return RESRC_DELAYTO;
   else {
      printf("unknown resource %s\n", name);
      return -1;
   }
}
```

Figure 6.8: Parts of `GetResrcName` and `GetResrcType` in `mgmt/resrc.cc`

At this point the execution flow between non-aggregate and aggregate queries merges again and if the parsing succeeded (i.e., the return value is 0) the execution flow returns inside the function, where there is the invocation of the `StartTask` method.

6.3.1.3. Creation of a new task

The `StartTask` method initializes a new task and then sends a message with `MgmtClient::NetSend` method. The `NetSend` behaves differently depending of the execution mode: if MON is running on normal mode, it sends a network message to a peer, otherwise it creates an event that simulates the message.

The execution flow of the normal execution mode becomes impossible to follow at this point. The output of the logging messages was flooded with the management and membership messages, making too hard to understand the next step. We recovered the execution flow at a later point.

6.3.1.4. Event management

The event is an instance of the `MgmtEvent` class defined in `common/event.h`. The execution flow then goes back to the `DoLoop` method on `managed/main.cc`; this method fetches the event from the event queue and handles it to the `HandleEvent` method on `main.cc`. The `HandleEvent` method invokes the `HandleMgmtEvent` method, that is located in `main.cc` as well.

The `HandleMgmtEvent` method creates some daemons (instances of the `MgmtDaemon` class) and invokes the `MgmtDaemon::HandleMessage` method on each of them. The `HandleMessage` method, among other operations, initializes a session object (instance of `MgmtSession`) and invokes the `MgmtSession::InitSession` method.

6.3.1.5. Session management

In the `InitSession` method the simulation and real mode execution flows merge together. The `InitSession` method may either invoke the `MgmtSession::ScheduleRetransEvent` method or the `MgmtSession::SetSessionReady` method, as shown in Figure 6.9.

```
    if(children_list.size() > 0){
       sess_cmd.toByteArray(msg->data); //only difference is level
       msg->src_id = my_self.peer_id;   //and src id
       SendMessageToNeighbors(msg, &children_list); //just send
       ScheduleRetransEvent(init_task_seq);     //task_seq
    }else {
       //last_child.InitPeer(my_self);
       SetSessionReady();
    }
    return 0;
```

Figure 6.9: Final part of `InitSession` method in `session/mgmtsession.cc`

The just mentioned analysis of the two methods doesn't help in understanding the next step of the execution flow.

The execution flow for the two execution methods is resumed on the `MgmtSession::HandleMessage` method, where the message (an instance of `MgmtMessage` defined in the `session/mgmtmsg.h` file) is checked against the management message macros. The `session/mgmtmsg.h` file contains the definition of those macros, that begin with `MGMT_MSG_` and identify the type of message; in the case of a MON query, the management message has a direct correlation with the names returned from the `ResrcSensor::GetResrcName` method (see Figure 6.10 and Figure 6.8).

```
[…] switch(msg->msg_type){
     case MGMT_MSG_NEWSESS:
        return HandleNewSessMessage(msg, sender);
     case MGMT_MSG_NEWSESSOK:
        return HandleNewSessOKMessage(msg, sender);
     case MGMT_MSG_PRUNE:
        return HandlePruneMessage(msg, sender);
     case MGMT_MSG_STOP:
        return HandleStopMessage(msg, sender);
     case MGMT_MSG_REFRESH:
        return HandleRefreshMessage(msg, sender);
     case MGMT_MSG_REFRESHOK:
        return HandleRefreshOKMessage(msg, sender);
     case MGMT_MSG_SESSCHANGE:
        return HandleSessChangeMessage(msg, sender);
     case MGMT_MSG_CMDFILTER:
     case MGMT_MSG_CMDTOPOLOGY:
     case MGMT_MSG_CMDCOUNT:
     case MGMT_MSG_CMDDEPTH:
     case MGMT_MSG_CMDAVG:
     case MGMT_MSG_CMDHISTO2:
     case MGMT_MSG_CMDSEL:
     case MGMT_MSG_CMDTOPK:
     case MGMT_MSG_CMDSYS:
     case MGMT_MSG_DATA:
     return HandleTaskMessage(msg, sender); […]
```

Figure 6.10: Part of `HandleMessage` method showing the management message macros

6.3.1.6. The status and management tasks

The `HandleMessage` method invokes the `HandleTaskMessage` method, that deletes the current status task and creates a new task. These steps are visible in the parts 1 and 2 respectively of the Figure 6.11.

The deletion of the current status task

```
//newer command message, old task is deleted
if(msg->msg_type == MGMT_MSG_CMDFILTER ||
                msg->msg_type == MGMT_MSG_CMDTOPOLOGY ||
                msg->msg_type == MGMT_MSG_CMDCOUNT ||
                msg->msg_type == MGMT_MSG_CMDAVG||
                msg->msg_type == MGMT_MSG_CMDHISTO2||
                msg->msg_type == MGMT_MSG_CMDSEL ||
                msg->msg_type == MGMT_MSG_CMDTOPK||
                msg->msg_type == MGMT_MSG_CMDSYS||
                msg->msg_type == MGMT_MSG_CMDDEPTH ) {
   delete status_task;
   status_task = NULL;
}else return 0;
```

The creation of a new status task

```
//here create new tasks
if(msg->msg_type == MGMT_MSG_CMDTOPOLOGY)
   status_task = new MgmtTaskTopology(this);
else if(msg->msg_type == MGMT_MSG_CMDCOUNT)
   status_task = new MgmtTaskCount(this);
else if(msg->msg_type == MGMT_MSG_CMDDEPTH)
   status_task = new MgmtTaskDepth(this);
else if(msg->msg_type == MGMT_MSG_CMDAVG)
   status_task = new MgmtTaskAvg(this);
else if(msg->msg_type == MGMT_MSG_CMDSEL)
   status_task = new MgmtTaskSel(this);
else if(msg->msg_type == MGMT_MSG_CMDTOPK)
   status_task = new MgmtTaskTopK(this);
else if(msg->msg_type == MGMT_MSG_CMDSYS)
   status_task = new MgmtTaskSys(this);
 [...]
```

Figure 6.11: The deletion and recreation of a status task in `MgmtSession::HandleTaskMessage` method

For the non-aggregate queries, the status task triggers the creation of an instance of `MgmtTaskSel` class, defined in `session/mgmttask.h` file. This class, like all the other classes whose name begins with `MgmtTask`, extends the `MgmtTask` class. The definition of `MgmtTask` presents some virtual methods, shown in Figure 6.12.

```
virtual int InitTask(char*data, int data_len) = 0; //e.g.,
  create filter
virtual char* GetTaskCmdData() = 0;  //for purpose of re-send
  data
virtual int GetTaskCmdDataLen() = 0;
virtual int LocalExec() = 0;
virtual int ReceiveData(const char* data, int len, int i) = 0;
virtual int UndoData(char*data, int len, int i){ return -1; }
virtual int AggregateData() = 0;
```

Figure 6.12: Virtual methods of the definition of `MgmtTask` class

The `MgmtTaskSel` class and all the other classes whose name begins with `MgmtTask` implement those methods, as shown in Figure 6.13.

```
class MgmtTaskSel: public MgmtTask {
public:
   char sel_desc[MAX_DATA_LENGTH];
   int sel_desc_len;
   SelData sel_data;
   MgmtTaskSel(MgmtSession* s): MgmtTask(s){};
   int InitTask(char*data, int data_len);
   char*GetTaskCmdData() {return sel_desc;}
   int GetTaskCmdDataLen(){return sel_desc_len;}
   int LocalExec();
   int ReceiveData(const char*data, int len, int i);
   //int UndoData(char*data, int len, int i);
   int AggregateData();
};
```

Figure 6.13: The definition of `MgmtTaskSel` class in `session/mgmttask.cc`

6.3.1.7. Query execution

After the creation of a new status task, there's invocation of the `MgmtTask::DoTask` method. This method exploits the polymorphism of the classes invoking the `InitTask` method. As result, in the case of a non-aggregate query the status task is an instance of `MgmtTaskSel`, therefore the `MgmtTaskSel::InitTask` method is executed. The code that represents the execution of the specific command is in the `LocalExec` method. If the query is an aggregate query, the `AggregateData` and `SendData` methods are exectuted and the query is completed. The Figure 6.14 contains the code of `MgmtTask::DoTask` related to the description of the last part of the execution flow.

```
//5. execute task
if(task_cmd.where_clause.eval() == 1) LocalExec();

if(task_children.size() == 0){
   task_state = TASK_STAT_READY;
   AggregateData();
   SendData();
}else task_state = TASK_STAT_INIT;
return 0;
```

Figure 6.14: The execution of the query code in `MgmtTask::DoTask` method

6.3.1.8. Issues in code analysis

The process of reconstructing the MON code execution flow and of locating the places where to change the code to support new keyword in the query language required much more time than planned. The reason of the delay lies in the following MON code design: the keywords included in the `keyw_table` are not all the allowed keywords of the MON query language, but a small subset of them. The TK_NAME macro includes the token associated to the MON parameters (among the parameters: CPU and memory usage, free disk space) and any other invalid token.

Consequently, the syntactic parsing is split into more than one location, a very important and uncommon design decision that is not documented.

6.3.2. MON binaries on PlanetLab

The MON instances run on the `uiuc_mon` slice of PlanetLab. The mon2 folder of each node (the full path is: ~/mon2/) contains all the MON files. The most important ones are the following:

- `remote_start_mon2`, a bash script that checks if the MON instance is running on the node and, if it's not, starts it;
- `managed`, the MON executable;
- `memberlist`, the list of MON nodes. Its syntax is shown in Figure 6.15;
- `myid`, a file that contains the MON nodeid;
- `pid.managed`, a file where `managed` writes its current PID (Process ID).

Before the creation of the MON refresh script described in Section 6.4, the MON management was performed with a Vxargs ssh script that executed the `remote_start_mon2` script.

```
#format is: id,ip,port
1234,192.17.239.251,6025
1235,192.17.239.253,6025
1236,195.37.16.97,6025
1237,128.112.139.80,6025
1238,219.243.206.124,6025
```

Figure 6.15: The first entries of the `memberlist` file

6.3.3. MON web interface

MON has a web interface [MONCGI] to easily submit queries to the MON instances on PlanetLab. The submission of a query on the web interface triggers the execution of the `batchclient` binary executable, that sends the query by default to

the `planetlab2.cs.uiuc.edu` node, therefore if that node is unavailable the web interface doesn't work unless a different server is specified in the query.

The source code of `batchclient` is unavailable, therefore it's impossible to change it to make a T-MON web interface. The code of the cgi script is not commented and consists, for the vast majority, of code not necessary for the purpose of submitting MON queries and displaying the results.

6.4. MON refresh

The PlanetLab nodes, as shown earlier in this chapter, exhibit low reliability. Therefore the number of running MON instances tends to decrease with time. It is important to manually perform the refresh of MON nodes at least once a week, but people often forget to do it or do it without regularity. The current script does not take into account the situations following a node reinstallation, that entails the deletion of the MON files. It is important to copy again the MON files on the node, but the current script was not able to do it.

The MON refresh procedure consists of an automated periodic execution of a refresh script. We created a cron job entry on `miami.cs.uiuc.edu` server to execute a bash script, uiuc_mon-cronscript.sh, shown in Figure 6.16.

```
#!/bin/bash
export TERM=dumb
#echo "Starting vxargs"
/home/ccampeg2/.mon2-refresh/vxargs-cron --plain -y -a
     /home/ccampeg2/.mon2/memberlist-ip --timeout=460 -P 16 -o
     /tmp/vxargs/result-cron /home/ccampeg2/.mon2-
     refresh/refresh_mon.sh {}
#echo "vxargs completed"
```

Figure 6.16: The `uiuc_mon-cronscript.sh` script

The purpose of the cron script is to decouple the cron job from the execution of the refresh, providing a better extendibility and easier management of the script. As seen in Figure 6.16, the cron job requires the full path of the script.

```
#!/bin/bash
export TERM=dumb
#echo "Start of refresh_mon.sh"


result=`/usr/bin/ssh -l uiuc_mon -i id_rsa-open -o BatchMode=yes -
     o StrictHostKeyChecking=no $1 "ps aux | grep -v grep | grep -
     c managed" `
#echo "result collected"
if [[ $result == 0 ]]
  then
    # MON is not running.
    # Let's see if the files are still there
    # then run managed again
    #echo 'rsync'
    /usr/bin/rsync -Cav -e "ssh -l uiuc_mon -i id_rsa-open"
     /home/ccampeg2/.mon2/ $1:~/mon2/
    COMMAND="cd ~/mon2;./remote_start_mon2 $1"
    #echo $COMMAND
    /usr/bin/ssh -l uiuc_mon -i id_rsa-open -o BatchMode=yes -o
     StrictHostKeyChecking=no $1 "${COMMAND}"
fi
```

Figure 6.17: The refresh_mon.sh script

The MON refreshing script, refresh_mon.sh (shown in Figure 6.17), uses Vxargs in atomic mode to perform the following algorithm: an ssh connection checks if MON process is running on the node. If the result is zero the node has not the managed process running, therefore two commands are executed: an rsync command that copies the content of the mon2/ folder to the node and then a ssh connection that executes MON. The use of rsync avoids the check of the presence of the MON files on the node making the script simpler and easier to understand; it also saves bandwidth, transferring the files only when needed.

Since cron runs in non-interactive mode, the only possible implementation was to create a passwordless private key to use for the Vxargs script. We stored the key on miami.cs.uiuc.edu server and we set its permissions in a way that only the

owner is allowed to access it, making the MON refresh procedure compliant with the security policy of Computer Science Department at the University of Illinois.

The list of PlanetLab nodes running MON instances can be seen on CoMon website [COMON] on the "slice-centric" section and selecting `uiuc_mon` as slice.

7. The new parameters for T-MON query language

This chapter describes the procedure that led to the design of the T-MON query language. The first step was to interview the students doing their research on the TEEVE project to see what parameters they would be able to monitor. At the end of the interview about the parameters, we designed and implemented a T-MON query language on the MON code.

7.1. Study of TEEVE parameters

We performed several interviews with the people actively working on the TEEVE project, to better understand the limits of TEEVE and to see what were the functions of the system that they needed. The list of functions (that eventually we mapped to parameters) included an indication of priority, that we took into account.

The gateway is the most important node of the system; because of that and, as a consequence, because it stores many parameters, it is the TEEVE component more likely to receive many requests. The gateway parameters to monitor, according to the people working on TEEVE project, were the following:

- Data traffic between gateways, i.e., the gateway to gateway stream data rate on the WAN network;
- Delay between gateways, i.e., the network routing delay from one gateway to the other gateway;

When communicating with other gateways, TEEVE gateways wrap all the streams into one single stream. The data traffic and the delay between gateways

should allow to extract data about single streams, by allowing for a greater granularity. The values for single streams are easily computable once single streams values are available. Another request is to be able to distinguish between audio and video streams.

- number of streams, distinguishing between inbound and outbound streams;
- packet loss rate, both inside the LAN and between gateways;
- CPU and memory usage;
- available bandwidth, i.e., the effective bandwidth available for the gateway to gateway communication. The available bandwidth varies depending to the network conditions.

The parameters for cameras, displays and trigger useful to measure by the members of the TEEVE project were the following:

- CPU and memory usage of each node of TEEVE;
- frame rate of each camera, i.e., the actual output frame rate;
- frame rate of each renderer, i.e., the displaying frame rate of the renderer;
- internal traffic, i.e., the traffic generated to and from the network;
- Basic status information about the trigger (the ability to know if it's online or offline would be sufficient).

7.1.1. Final considerations

Many of the listed parameters are already collected from the node applications in TEEVE, so they can be implemented in the T-MON query language.

Regarding the gateway, the parameters with low priority were the following:

- capability to measure the audio stream characteristics (bandwidth and delay), as audio support is in the early stage of development;
- available bandwidth, because it's hard to measure it from the gateway point of view and it's not currently measured by the gateway. Other tools have also the capability to measure it.

A person of the TEEVE group also asked to display the network topology, but it was a low-priority request and it would have been too complicated with respect to the benefits provided, as the TEEVE network topology has few changes over time. Regarding the other nodes, we defined the parameter about trigger as low-priority. As T-MON design decision, we did not consider the low-priority parameters for inclusion in the T-MON query language.

7.2. T-MON query language

The MON query language has two main types of query: *aggregate data* and *non-aggregate data*. Considering the list of TEEVE parameters to include in the query language and the usage scenario of T-MON, the query language was of non-aggregate data query type. The usefulness of the aggregate data is lower when the number of nodes is in the range of some tens. The T-MON typical usage scenario is to monitor the TEEVE system, so the where clause must be present to force the user to monitor a small set of parameters for each query.

7.2.1. The T-MON parameters

The Figure 7.1 shows the proposed T-MON query language.

```
select <resource> where <condition>

<resource> keywords:          <condition> keywords:
camera                        localnode
renderer                      site
gateway                       objectid
framerate                     framerate
lantraffic                    lossrate
delay                         value
lossrate                      type
```

Figure 7.1: Proposed keywords for T-MON query language

The resource keywords represent the parameters as described before: camera and renderer represents a camera and renderer computer, respectively; a gateway represents a gateway computer. The framerate represents either the camera or the renderer frame rate, the lantraffic represents the traffic of the node, delay represents the gateway to gateway network delay; the lossrate represents the amount of packets lost in the WAN link.

The meaning of the keywords of the where clause (condition) is the following: the localnode is meant to refer to a specific node. An example of condition is the following: `localnode = 128.174.247.67`. That would help to query parameters of a specific machine, like the `select lantraffic where localnode = 128.174.247.67` query.

The meaning of the site keyword is similar to the one of localnode, but with a greater span, because is meant to refer to a TEEVE site, i.e., to query the nodes of a specific location. A sample query using the site keyword would be the following: `select camera where site = uiuc`. That query would return the working cameras on the uiuc TEEVE site.

The objectid keyword has a more general meaning that depends by the keyword used in the were clause. This decision comes from the observation that a query language cannot include all the parameters used on TEEVE, but with objectid it's possible to extend the query language with minor improvements without redefining the grammar. From the coding point of view, the modification is easy, as we have to apply the changes to one method. A sample query using the objectid keyword might be the following: `select gateway where objectid = stream3 and lossrate > 2`, That query returns the gateways, if any, that have the stream3 working and have a lossrate greater than 2 percent.

The framerate keyword represents the frame rate of the camera or the renderer. We have to place it on the where clause keywords, other than in the select clause keyword, because it is a parameter that can be used for different situations. When the

framerate is in the select clause of the query, it has the following meaning: “select the framerates of nodes where <where_condition>”; when it’s used in the where clause of the query, it has the following meaning: “Among the nodes that have a framerate (equal to, or greater than, or less than) a certain value, return (select) the <resource>”. The same deductions apply for the keyword lossrate, that is also among the keywords of the select clause of the query.

The meaning of the value keyword is to provide the filter capability over the value of the keyword of the select clause. This modifies the semantic of the query language, as we can use any select clause keyword to set a condition on the where clause, but it provides benefits in terms of flexibility. This is different from the use of the select clause parameters for the where clause parameters, because the query with the value keyword has a strict query semantic over the value of the same parameter. With the value condition, queries like the following one are allowed: `select delay where value > 3`.

The meaning of the type keyword is to provide the use of the new T-MON query parameters with the MON keywords. The allowed values for the value keyword are the T-MON resource keywords. A sample query using the type keyword is the following: `select freemem where type = camera`.

7.2.2. MON architectural limits

MON performs the evaluation of the select clause in a similar way to the evaluation of the where clause. In particular, all the tokens are valid either in the select clause and in the where clause. This behavior is undesired as, according to the design illustrated, few parameters can be valid in both clauses, while the majority can’t.

The attempts to differentiate the evaluation of the where clause from the evaluation of the select clause failed because of a design limitation of the MON architecture, that is based on Protocol Plug-in Framework.

The possibility to catch the query at the semantic level is minimal, as the semantic evaluation is spread in several parts of the code, so that it is impossible to evaluate the entire query string once parsed. This happens because the token reader removes the substring preceding the current token, i.e., in case of a semantically valid query the incoming buffer (that contains the query string) at the end of the parsing is empty.

7.2.3. The final T-MON language

Because of the MON code limits and because of the high complexity in adding a new keyword to it, we selected the parameters shown in Figure 7.2:

```
camera
renderer
gateway
framerate
lantraffic
delay
lossrate
site
```

Figure 7.2: Selected keywords for T-MON query language

These parameters can appear in the select clause and/or in the where clause.

We did not implement the value and the type keywords during the first phase of the development because of the greater complexity in the implementation with respect to the implementation of the other T-MON query parameters.

7.2.4. Formal analysis of the T-MON query language

This section describes the formal analysis of the T-MON query language. The T-MON EBNF (Extended Backus–Naur Form) grammar is shown in Figure 7.3.

The EBNF grammar provides a standard formal representation of a programming language and consists of four sections: VT, that is the list of terminal

symbols, VN, that is the list of non terminal symbols, S is the goal of the parsing and P is the list of productions.

```
VT = {1,2,3,4,5,6,7,8,9,0,<,>,=,}
VN = { <query>, <params>, <param>, <boolexpr>, <shortexpr>,
     <longexpr>, <op>, <eq>, <val>, <digit>, <nzdigit>, <monparam>
     }
S    = <query>
P    = {
<query>         ::= select <params> where <boolexpr>  EOL
<params>        ::= <param> {“,” <param>}
<param>         ::= camera | renderer | gateway | framerate |
     lantraffic | delay | lossrate | site | <monparam>
<boolexpr>      ::= <shortexpr> | <longexpr>
<shortexpr>     :: = <param> <op> <val> | “(” <shortexpr> “)”
<longexpr>      ::= <shortexpr> and <shortexpr> { and <shortexpr> }
<op>            ::= >{<eq>} | <{<eq>} | <eq>
<eq>            ::= =
<val>           ::= <digit> | <nzdigit> { <digit> }
<digit>         ::= 0 | <nzdigit>
<nzdigit>       ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<monparam>      :: = busycpu | freemem  | diskusage  | freedisk |
     uptime
 }
```

Figure 7.3: EBNF grammar of T-MON query language

According to the Chomsky hierarchy [WPCHO], T-MON language is a type 2 language with a context-free grammar. The language, albeit context-free, is not regular because there is the self-embedding in the `shortexpr` production. We intentionally left the self-embedding in the T-MON query language to preserve MON querying capabilities, as TEEVE nodes may use the query capabilities of MON to get more node information.

The empty string does not belong to the language and on the syntactic level it cannot be generated, even from the goal.

7.3. Inclusion of TEEVE parameters in T-MON query language

This section describes the process of adding a new keyword to the MON query language. The added keyword is the following string: test. The macros associated to it are `MGMT_MSG_TEST` and `RESRC_TEST`.

We defined the `MGMT_MSG_TEST` macro on `mgmtmsg.h` and on `mgmtmsg.cc` files, as shown in Figure 7.4.

`mgmtmsg.h`

```
#define MGMT_MSG_TEST 1201
```

`mgmtmsg.cc`

```
const char* MgmtMessage::MsgName(int
     type)
{
   switch(type){
     case MGMT_MSG_ECHO:
          return "ECHO"; […]
     case MGMT_MSG_TEST:
          return "TEST"; […]
```

Figure 7.4: Additional code in `mgmtmsg.h` and `mgmmsg.cc` files

All the `MGMT_MSG_` macros related to T-MON parameters have value greater than 1200.

We also have to add the macro has also in the `mgmtsession.cc` and `mgmtcmd.cc` files (as shown in Figure 7.5 and Figure 7.6, respectively).

`mgmtsession.cc - MgmtSession::HandleTaskMessage method`

```
if(msg->msg_type == MGMT_MSG_CMDFILTER ||
                    msg->msg_type == MGMT_MSG_CMDTOPOLOGY ||
                    […]
                    msg->msg_type == MGMT_MSG_TEST ) { […]
```

`mgmtsession.cc - MgmtSession::HandleMessage method`

```
switch(msg->msg_type){ […]
    case MGMT_MSG_TEST:
       return HandleTaskMessage(msg, sender); […]
```

Figure 7.5: Additional code in `mgmtsession.cc` file

```
mgmtcmd.cc - MgmtCommand::toByteArray method

[…] switch(cmd_type){ […]
      case MGMT_MSG_TEST:
        sel_data.toByteArray(p); p += sel_data.GetByteLength();
        break; […]
```

```
mgmtcmd.cc - MgmtCommand::GetByteLength method

[…] switch(cmd_type){ […]
      case MGMT_MSG_TEST:
        total_len += sys_data.GetByteLength();
        break; […]
```

Figure 7.6: Additional code in `mgmtcmd.cc` file

We have to add a new class definition, `MgmtTaskTest`, in the `mgmttask.h` file. The class inherits from the `MgmtTask` class and implements its virtual methods, whose most significant are the `InitTask`, `LocalExec` and `AggregateData`. The implementation of the class in the `mgmttask.cc` file must adds the task initialization code in the `InitTask` method, the code executed to serve the query and provide a result in the `LocalExec` method, and the aggregation function in the `AggregateData` method.

Following the creation of a `MgmtTaskTest` class, we have to add a reference to that class in the `mgmtsession.cc` file. The invocation, shown in Figure 7.7, exploits the polymorphism of the `MgmtTask` class, invoking the corresponding methods of the `MgmtTaskTest` class.

```
mgmtsession.cc - MgmtSession::HandleTaskMessage
    if(msg->msg_type == MGMT_MSG_CMDFILTER ||
                        msg->msg_type == MGMT_MSG_CMDTOPOLOGY || […]
                        msg->msg_type == MGMT_MSG_TEST ) {
          delete status_task;
          status_task = NULL;
       }else return 0; […]

    else if(msg->msg_type == MGMT_MSG_TEST)
          status_task = new MgmtTaskTest(this); […]
```

Figure 7.7: Adding the invocation to `MgmtTaskTest` class in `mgmtsession.cc`

7.3.1.1. The inclusion of the resource

The following two steps, shown on Figure 7.8, consist in the definition of `RESRC_TEST` macro in the `resrc.h` file and in the inclusion of that macro to `mgmttask.cc` file.

`resrc.h`

```
#define RESRC_TEST  701
```

`mgmttask.cc` - `MgmtTask::GetResrcStrValue` method

```
switch(resrc_type){
      case RESRC_VERSION: […]
      case RESRC_TEST:
         sprintf(tmpbuf, "%.2f", value);
         break;
```

Figure 7.8: Inclusion of `RESRC_TEST` macro

All the T-MON resource macros have a value ranging from 701 to 800.

Finally, we have to add the `RESRC_TEST` macro to the `resrc.cc` file, as shown in Figure 7.9.

```
resrc.cc - ResrcSensor::GetResrcName method
  switch(resrc_type){
     case RESRC_TEST:
          p = "TEST"; break;

resrc.cc - ResrcSensor::GetResrcType
   else if(!strncmp(name, "test", 4)) return RESRC_TEST;
```

Figure 7.9: Additional code in the `resrc.cc` file

The process of adding one parameter to the query language is verbose and increase the time required to perform it; such code structure also harms the extendibility of the code.

7.4. User view

The T-MON system is for people working on the TEEVE project, that have a deep scientific understanding in the field of Computer Science issued. The SQL-like syntax of the query is not therefore an obstacle to the use of T-MON.

The typical usage scenario involves a user that periodically performs queries. Since TEEVE system is continuously developed, the use of T-MON helps to check the results of new code allowing a selective monitoring of a single type of node. Imagine for example a change in the camera code. We can deploy the new version of the code to the cameras while the rest of TEEVE has the old version of the code running. With T-MON a selective monitoring of all the cameras is easier and more flexible. We can also configure a script to periodically send T-MON queries and save their result for further offline studies.

8. T-MON Architecture Framework

This chapter presents the T-MON architecture and design study. It explains the evaluated design options and the reasons behind the design choice. The resulting design is not part of the T-MON architecture due to the problems experienced with the MON code and PlanetLab package repository described in Chapter 6.

Initially we planned to split the development into two phases: the first phase involved the creation of a static version, where the TEEVE system is simulated using trace files or ad-hoc created log files to test the interactions between the T-MON components and evaluate the overhead. The second phase (that was not possible to implement because of the issues with the project) would consist in the creation of the dynamic version, where the complete T-MON system is deployed to the PlanetLab network to measure the performances.

Since TEEVE runs on Windows machines and MON runs on Linux, the real TEEVE could not be used with T-MON; for this reason the plan included the creation of software TEEVE processes to simulate the interaction between each TEEVE component and T-MON. For this purpose, we studied the code of TEEVE gateway to see the functionalities offered by the gateway and to have a detailed view of what information (with particular focus to T-MON query parameters) the gateway stores.

8.1. Introduction to the architecture

The Figure 8.1 illustrates the TEEVE network architecture.

As illustrated in Chapter 2, the gateways are the only nodes that have access to the Internet2 (WAN) network and every connection between two locations is managed by the gateway. Cameras and renderers have high CPU and bandwidth requirements, while the gateway's critical resource is the bandwidth. We did not include the trigger in the analysis because it is not represented in the query language

and because a failure of the trigger is easily detectable as it fires a failure chain reaction on the other TEEVE nodes.

The main idea is that a process related to T-MON runs on the TEEVE nodes, next to the corresponding TEEVE process (i.e., TEEVE camera process on camera nodes, and so on). The people working on TEEVE project expressed interest for the ability to query the past values of parameters for a comparison. This would also allow them to have an activity log file at the end of each TEEVE session.

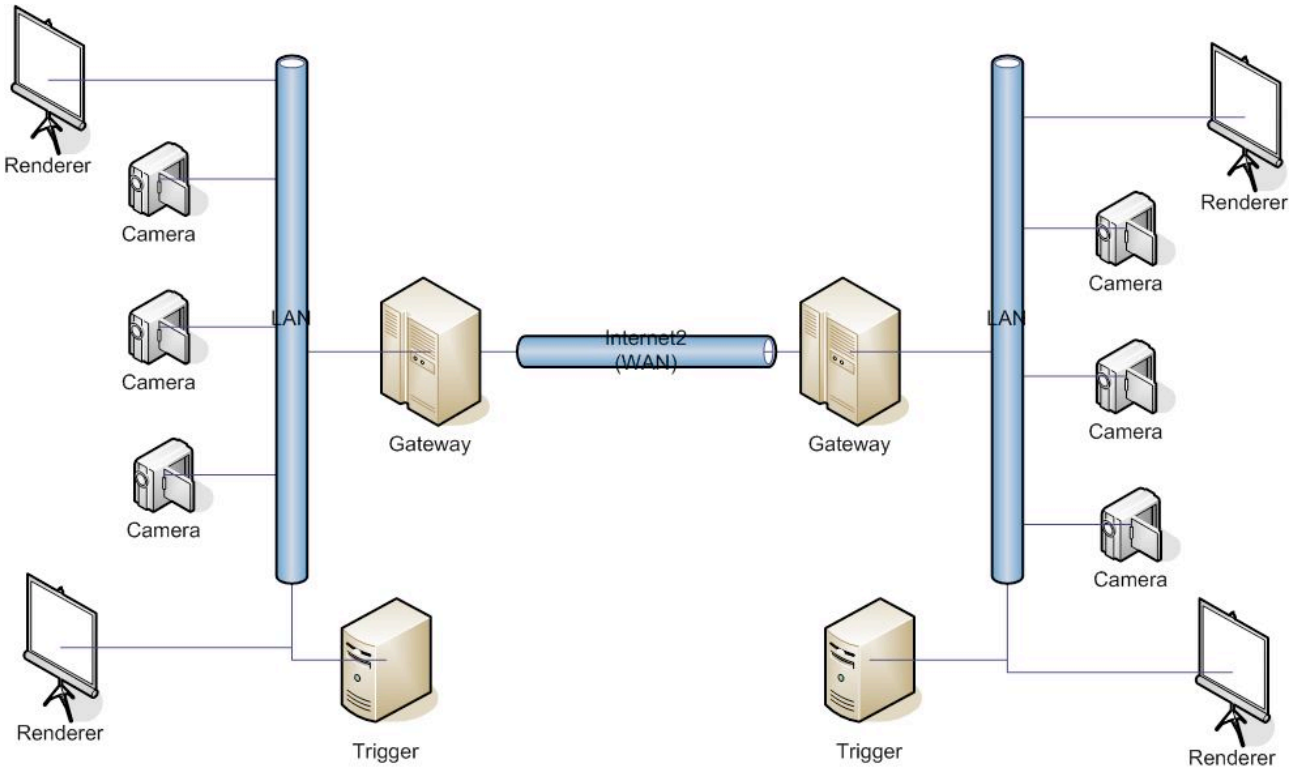


Figure 8.1: TEEVE network diagram

8.2. Initial design decisions

This section describes the first design decision regarding the T-MON architecture, that focus on the interaction between the T-MON and TEEVE processes on every node. We describe the solution regarding the interprocess communication method and a basic interaction model between the two processes.

8.2.1. A shared communication data structure

The first problem was the interfacing between the T-MON process and the corresponding TEEVE process. TEEVE process needs to provide the information to the T-MON process in a way similar to the producer-consumer model, guaranteeing the atomicity: if a T-MON process receives a query while an update for the parameter is happening, there may be problems of data consistency.

We considered the common interprocess communication mechanisms (such as shared memory, a shared file, local sockets) together with other data structures such as databases. The database software guarantees atomicity of the operation in a producer-consumer model, a characteristic that assures data integrity. The database software is also optimized for quick data retrieval: the answer to a database query is generally faster than information retrieval from a shared file, especially for a large data set (TEEVE produces very verbose logs). Moreover, the mapping of the T-MON queries to the database query is very simple because both systems use a SQL-based query language, providing a performance optimization.

The choice of database had to take into account the PlanetLab infrastructure, therefore we had to restrict it to database packages available on PlanetLab, i.e., MySQL [MYSQL] and PostgreSQL [PSQL]. We selected the MySQL database to use in T-MON after talks with graduate students in the Computer Science department of the University of Illinois with strong database-related research experience. Their common opinion was that PostgreSQL provides significantly better performances than MySQL for strongly optimized queries, but in the general usage scenario of T-MON the difference of query performance would have been minimal. Since PostgreSQL has more features than MySQL, it has also higher resource requirements. The MySQL version that we installed on PlanetLab and miami development server is the 4.1.

8.2.2. Overview of TEEVE and T-MON processes on a node

This section illustrates the basic interactions between TEEVE and T-MON processes on a node.

The T-MON process continuously sends messages for overlay and membership maintenance, consuming bandwidth and some CPU. This overhead, albeit small, may affect the performances of the cameras and renderers; even a small amount of resources can affect the real time requirements of those nodes. Therefore the T-MON process runs only on gateways, where the CPU usage isn't a main constraint, and each camera and renderer has a local T-MON agent, meant to use less resources than the T-MON process. On the gateway, together with the T-MON process, there is a T-MON data collection agent, whose role is to interact with the local agents. We will discuss the interaction between the local agents and the collection agent in the next sections. The interaction diagrams for the Gateway and camera-renderer are shown in Figure 8.2 and Figure 8.3, respectively. We may include the T-MON local agent on the gateway in the T-MON data collection agent.

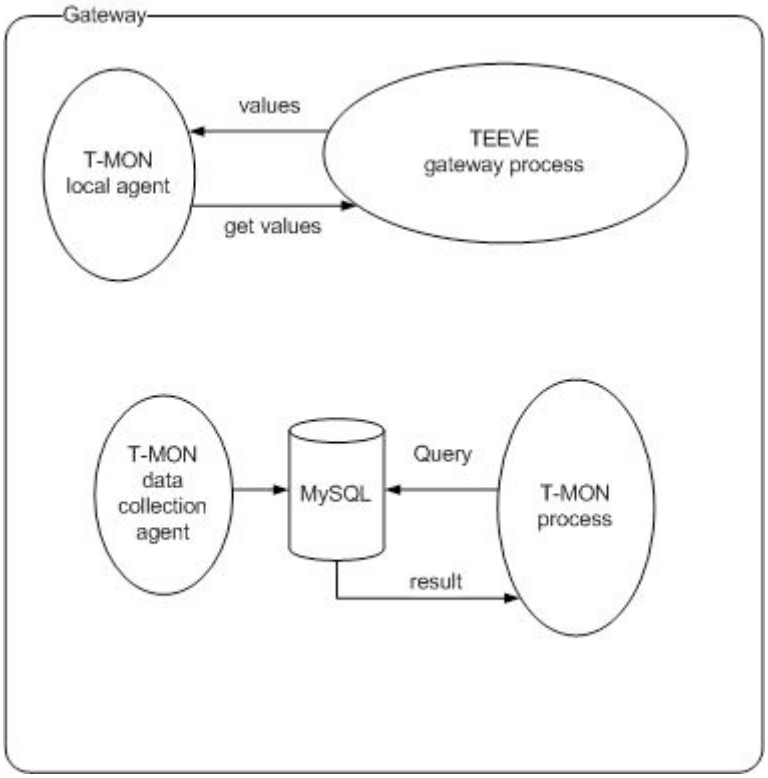


Figure 8.2: T-MON process and agents on the TEEVE gateway node

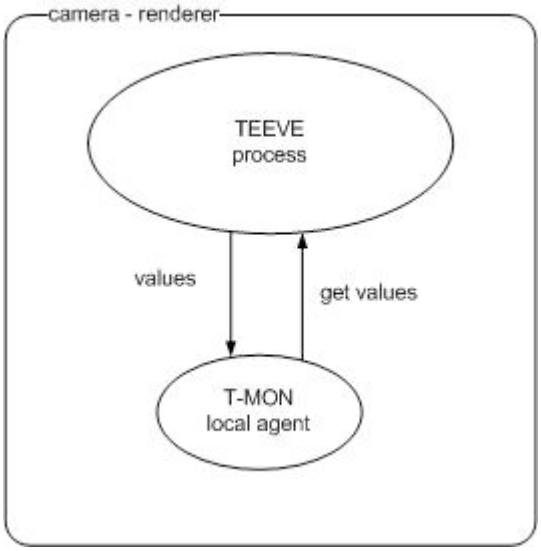


Figure 8.3: T-MON local agent model on the TEEVE camera and renderer nodes

The use of a T-MON data collection agent, instead of including its functionalities in the T-MON code, provides better performances, as T-MON is single threaded, and provides the functional separation between the processes: T-MON should focus on the querying functionality and not manage the data collection.

8.3. The communication model

There are two main communication semantics to be implemented between the local agents and the data collection agent: *push* (i.e., the local agents send periodically the data to the data collection agent) and *pull* (i.e., the data collection agent asks to the local agents the latest data).

While there are several ways to implement those two models in the T-MON scenario, we discuss only the most significant. With the push model the agents would periodically send the values to the data collection agent, generating a constant and predictable amount of traffic on the network; alternatively, local agents may decide to send data to the data collection agents only when the value of one or more parameters are outside a predefined range.

With the pull model the agents are simply remote interfaces between the data collection agents and the TEEVE process: the local agent receives the request from the data collection agent to provide one or more TEEVE parameter values, grabs them and sends back the reply with the values of the requested parameters. The pull model would provide an overhead advantage over the push model in case of a pure on demand approach, i.e., the data collection agent sends the request for the values of the TEEVE parameters only when the T-MON process receives a query. This approach increases the T-MON query response time, a parameter that we have to keep as low as possible, and is incompatible with the need to log the value of TEEVE parameters over time because the values would be sampled only when there is a T-MON query. As a consequence, the communication is based on the push model.

We can implement the push model in two ways, as written before: by sending the data to the data collection agent either regularly or when the values of the parameters are outside a predefined range. We chose the first approach, because there is currently no support in T-MON to query for parameter's anomalies. The time interval between the sending of the packet from the local agent to the data collection agent should ideally be as small as possible to have fresh data in the database, and as big as possible to minimize the network overhead.

The selected solution is a hybrid between those two needs: the local agent periodically polls the TEEVE process (T_{poll}) but it does not send the data to the data collection agent until it does not collect a certain number of samples (T_{send}, the time interval between two data send to the data collection agent). With this solution the sample interval in the database would be T_{poll} for the old data, providing more granularity in the reading of the values in case of a later examination, and at the same time the network overhead is reduced. The selected solution includes an additional scenario: if a T-MON query requires recent values, the data collection agent asks the local agents involved in the query to send immediately the TEEVE parameter values. This process increases the T-MON query response time, but the assumption is that on

a normal use those queries would be rare. Therefore we define the final solution as hybrid communication model, as there is a pull-like situation. Normally we consider valid a T-MON response to a query if the value is not older than few minutes.

8.3.1. Description of on-demand TEEVE parameters update

The sequence of the additional scenario, illustrated in Figure 8.4, is the following: when the T-MON agent receives a query, it asks the data collection agent to force the refresh of the required values (step 1 in the Figure). The data collection T-MON asks (steps 2, 3) the local agents to provide immediately the data; once the local agents receive the data, (step 4) the data collection agent updates the database. The data collection agent notifies T-MON process that the updated values are in the database (step 5) and the T-MON process reads the values from the database (steps 6 and 7, respectively).

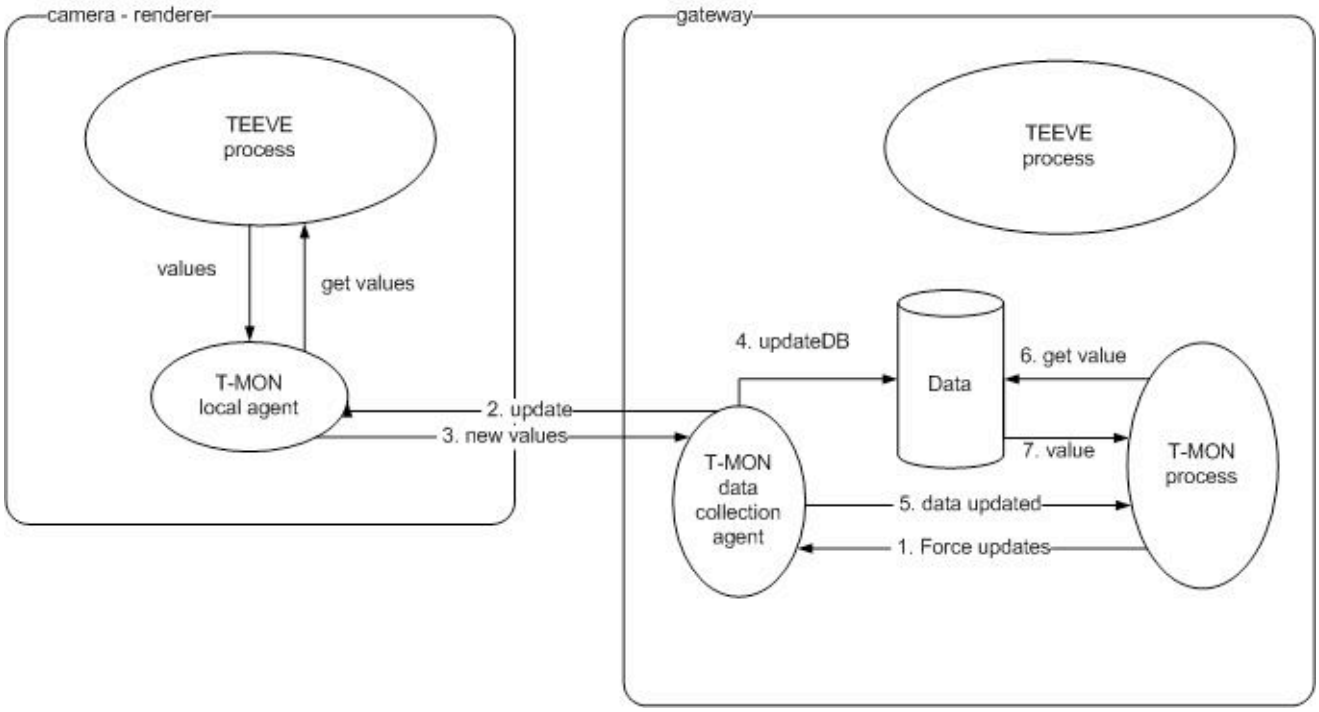


Figure 8.4: On-demand TEEVE parameters update

In the hybrid scheme the data collection agent is not required to send the acknowledgments of the receipt of the data to further save on bandwidth overhead. If the data collection agent does not receive a packet sent from the local agent and no

T-MON queries require the values included in the lost packet, the data collection agent can wait for the next data transmission form the local agent. Whenever T-MON receives a query that needs the missing values, it requests an on-demand parameters update. This approach, however, increase the T-MON query response time, so we require the use of the data receipt acknowledgments.

8.3.2. Additional remarks

To find the optimal values of the period, we need to perform the calibration of the data sending time intervals by testing different values, however we believe that values of about 30 seconds for T_{poll} and 90 seconds for T_{send} should be a good compromise between data freshness and communication overhead.

We also considered to install one database in each node, ideally replicating the union of data collection T-MON and local agents on all the camera and renderer nodes. This would have greatly reduced the network communication overhead, but would have added more CPU load to nodes with strong real-time computation requirements; moreover, the T-MON query response time would have increased as the T-MON process would have to perform a network database query instead of the local database query of the hybrid model.

8.4. Design of the T-MON data collection agent

This section describes the architecture of the T-MON data collection agent and shows two proposed designs for its implementation: the first one implements the pull model using the MySQL database as data storage and retrieval solution; the second one models a T-MON system where each node has its own local database and the T-MON process follows the on-demand approach for every query it receives.

Even if we explained the adopted solution and the reasons behind this choice in the previous sections of this chapter, we believe they are a reference implementation for the prosecution of the T-MON project.

In the first solution the gateway local agent is part of the data collection agent, while in the second solution the two agents are separated to maintain the separation between the local agents (that have a database) and the data collection agent (that hasn't a database).

Both solutions include the evaluation of anomalies, i.e., the comparison between the read TEEVE values and the neighborhood of the their typical value. The T-MON administrator can statically define both the typical values and neighborhood range; alternatively, they can change during time.

8.4.1. Description of the first solution

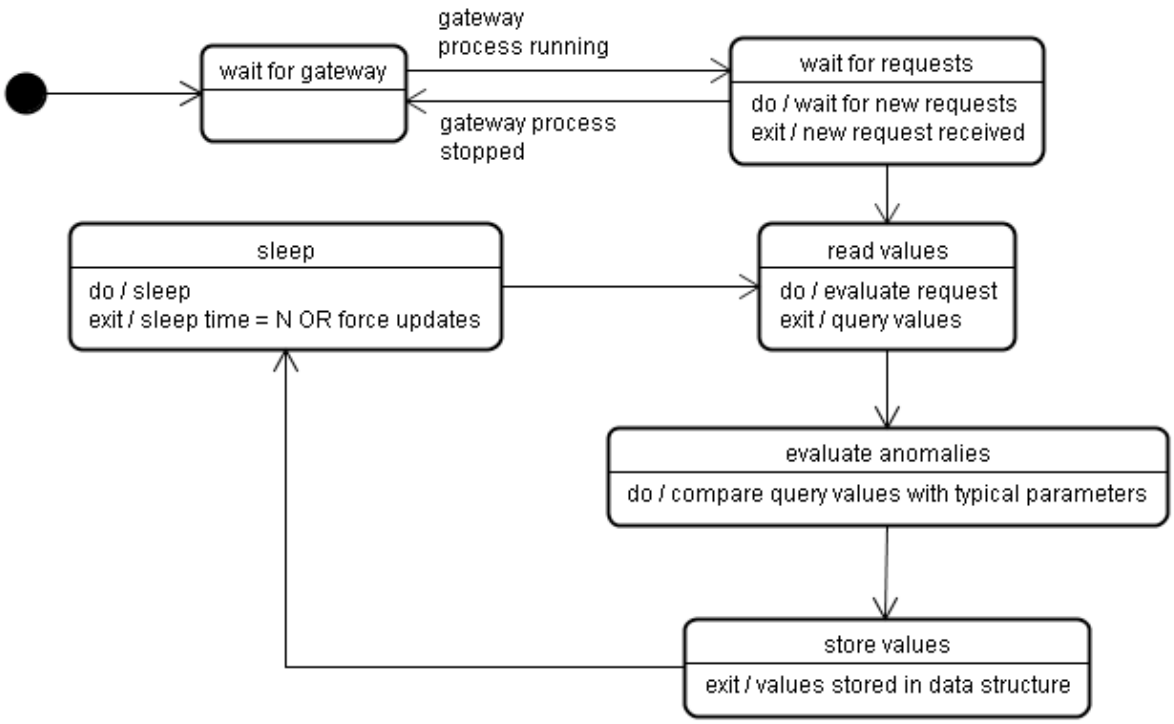


Figure 8.5: State chart of the first solution

In the first solution, the data collection agent includes the database, therefore at every T-MON query there is interaction between the T-MON process and the data collection agent, even if the intended implementation of this solution would make the data collection agent merely a wrapper to the database.

The Figure 8.5 displays the state chart of the first solution. The Figure 8.6 shows the state chart of the “read value” state of Figure 8.5.

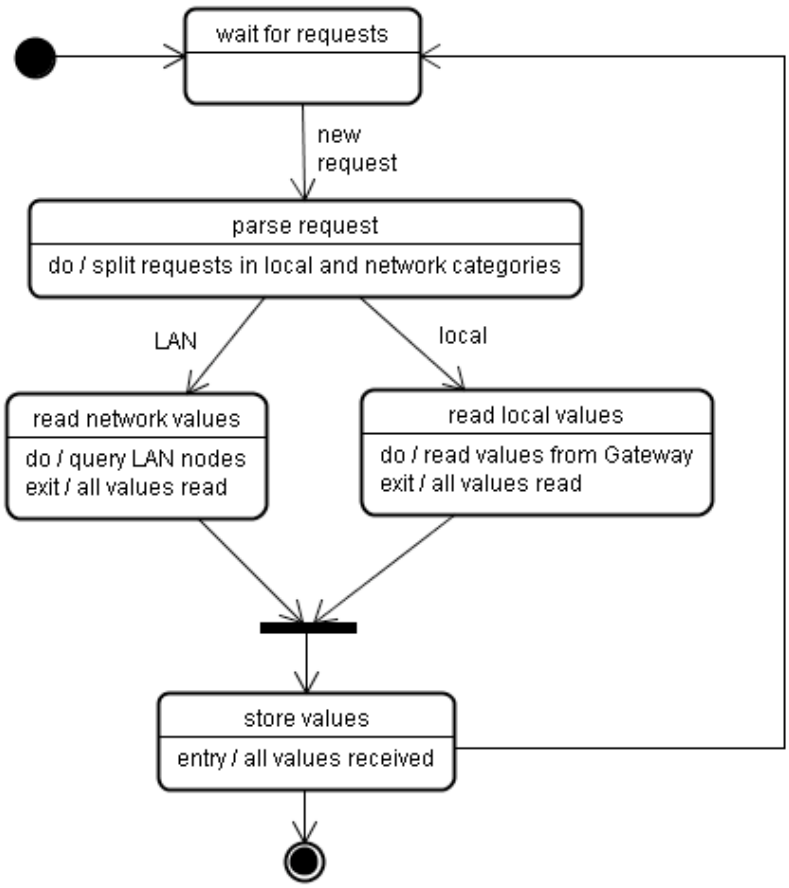


Figure 8.6: State chart of the "read value" state in the first solution

We highlight the on-demand behavior in Figure 8.6 as all the queries that not involve the gateway are forwarded to the respective local agents. In the first solution the database is mainly used to store the past values, as we have considered the option to include the capability to perform the T-MON query on past TEEVE parameters value. We abandoned this option because of the delays in the development of the project described in the previous chapters.

8.4.2. Description of the second solution

The Figure 8.7 shows the state diagram of the second solution. As previously described, it hasn’t a database and it has a hybrid mode semantic.

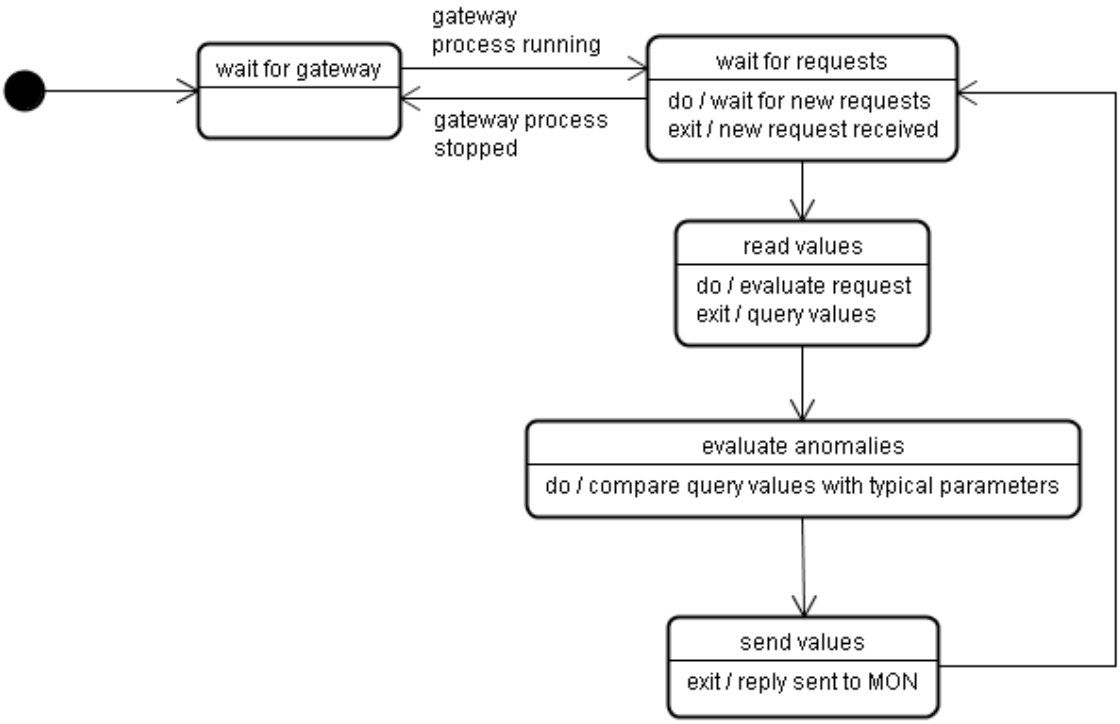


Figure 8.7: State chart of the second solution

The Figure 8.8 shows the state chart of the "wait for requests" state described in Figure 8.7. The difference with respect to the first solution (illustrated in Figure 8.6) is the final state, where the T-MON process receives the values, as there is no database (the first solution stores the solution in the database).

8.5. Simulated TEEVE data gathering

To have a simulation of the TEEVE system as close as possible to the real implementation, we asked to the most experienced people working in the TEEVE project to provide a trace file. In the past, trace files were recorded, but they have been eventually deleted. After some investigation those people reported that apparently that feature is not available in the current TEEVE release.

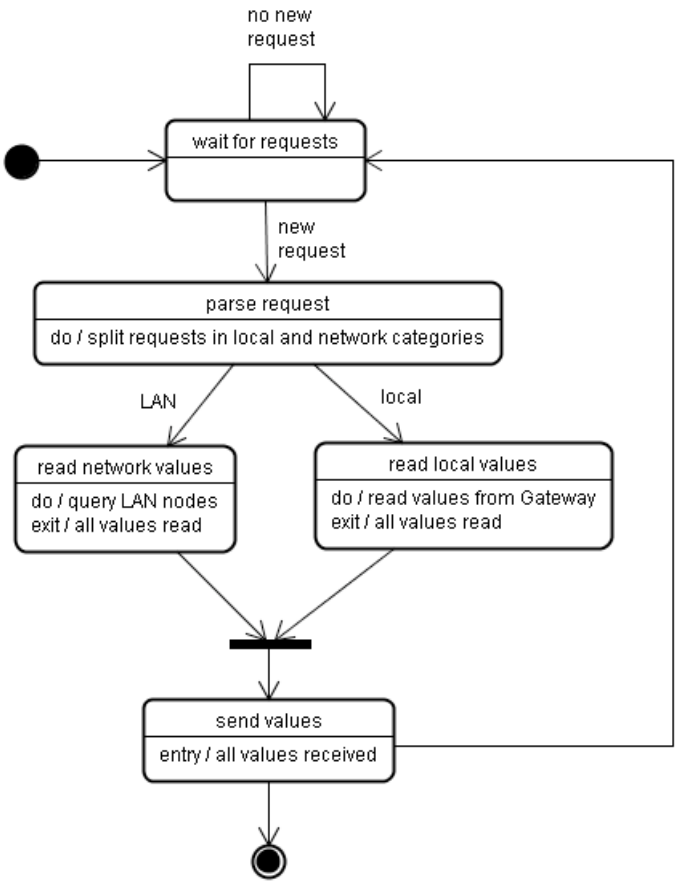


Figure 8.8: State chart of the "read value" state in the second solution

8.5.1. A possible sampling tuple

A proposed sampling tuple (i.e., a tuple that represents a TEEVE sample) is the following:

```
(fps, CPU, resolution, sample time, ON/OFF)
```

The `fps` field represent the frames per second (either of the camera or the renderer, depending from what data structure the tuple belongs to), the `CPU` represents the CPU usage as percentage, the `resolution` is the frame resolution, the `sample time` is a unique time identifier that refers to the sampling time of the tuple, `ON/OFF` refers to the status of the node.

The system can be designed so that we can use this tuple on different situations: for the database, for the communication between the T-MON local agents and the data collection agent, and as entry of the trace file.

There would be no need to implement conversion processes to convert the data from one format to another and vice versa, reducing the program complexity and increasing the flexibility.

8.6. User view

The TEEVE project users are the target of the planned implementation, usually Computer Science graduate students. For this reason they are expected to be familiar with the topic of distributed systems, the command line interface and the SQL query language.

The only user case is when the user performs a query: assuming the user knows the query language, he can query the TEEVE system either periodically to monitor the system or in case of a suspect of a malfunction, to locate the problem and perform a fix. The database stores the parameters values, so the user can access them when the TEEVE session ends and look at the parameter values history for feedback.

8.7. Complete T-MON system diagram

The Figure 8.9 shows the final T-MON architecture.

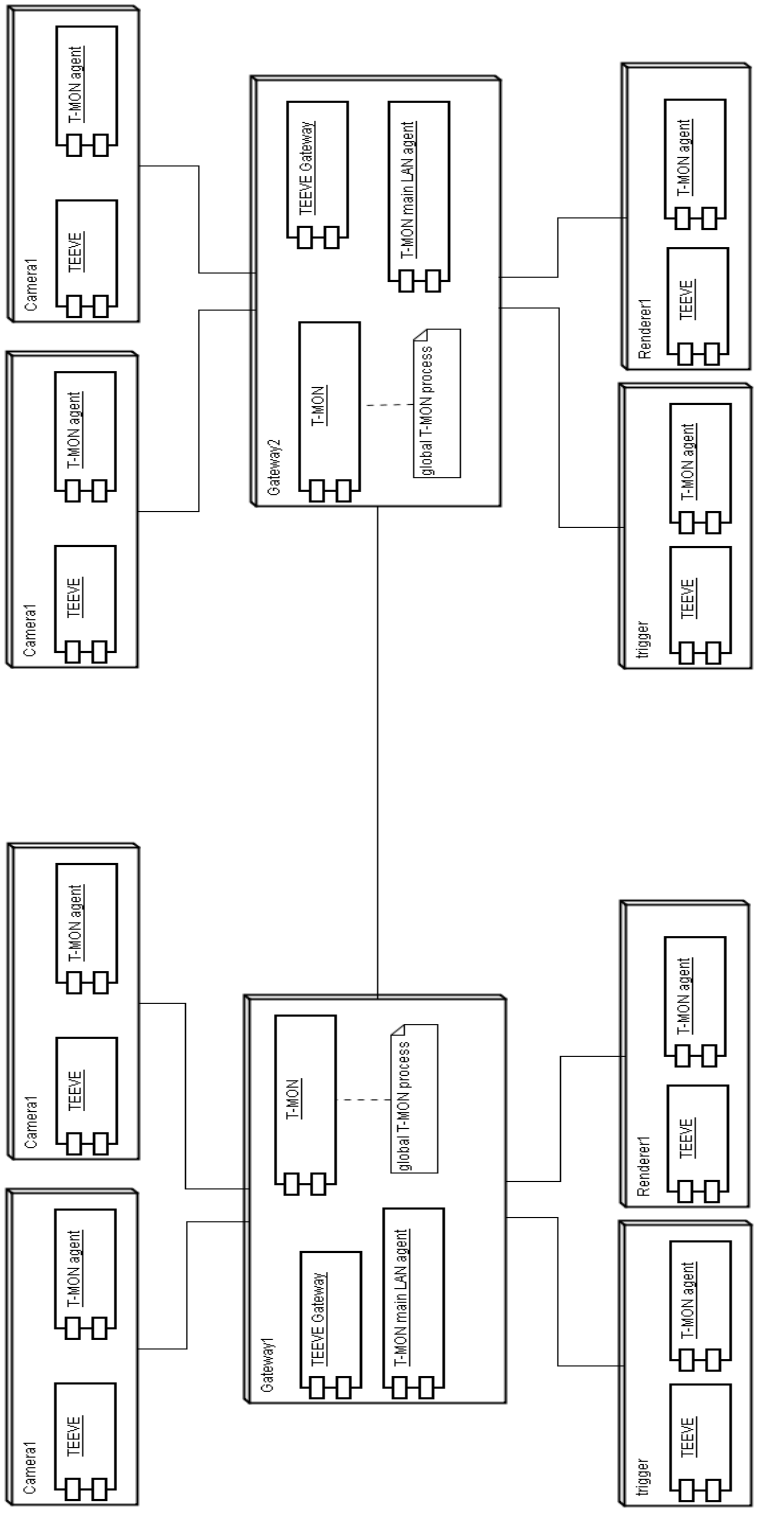


Figure 8.9: Complete T-MON system architecture diagram

The Figure 8.9 shows the complete T-MON system architecture, that includes all the design decisions described in this chapter, represented on a TEEVE system running on two different locations. Each rectangle represents a node and each line represents a network connection. The rectangles with two small rectangles on their left side represent the processes running on a node, while the dotted line connects a comment to the commented object.

Each node runs the corresponding TEEVE process and a T-MON process, except for the gateway, where there are two T-MON processes: the data collection agent (LAN agent), that mainly receives the data from the T-MON agents and stores them in the database, and the T-MON process, that runs only on the TEEVE gateways to minimize the overall overhead.

9. Validation of systems

The chapter discusses the validation of the MON and T-MON systems illustrated in this thesis. The T-MON evaluation refers only to the simulation, as the monclient application provided with MON appears to not work. However we describe the planned testbed scenario for T-MON.

9.1. MON Evaluation

This section evaluates MON: the tests compare the tree construction algorithms, the tree overlay metrics (coverage and response time), overlay tree reliability, and the quality of the results of MON queries. The evaluation tests ran both in simulation mode (i.e., the MON capability to simulate a real network usage) and on PlanetLab.

In the simulation mode, the total number of (simulated) nodes is `N` = 2000. Before the execution of each test there is a wait to let MON reach the steady state, and then some nodes are killed and an overlay tree is created. On the PlanetLab

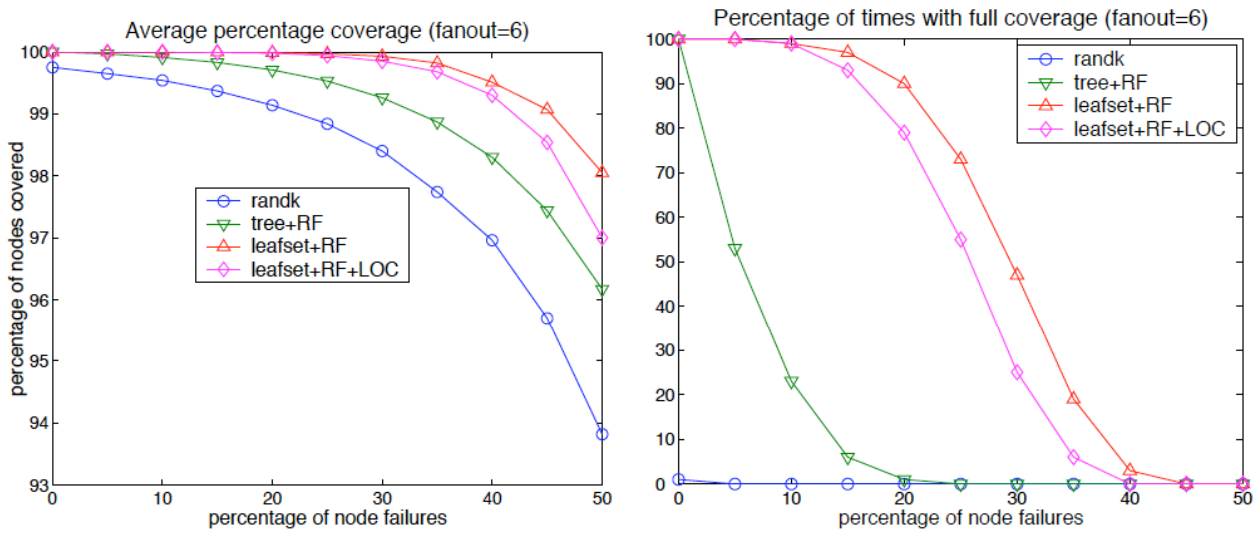
experiments, MON was deployed on about 320 nodes and created on-demand overlay trees using the different algorithms.

The first test ran in simulation mode. It evaluates the coverage and response time of the overlay tree created using the different algorithms, and the reliability of DAG overlays.

The Figure 9.1(a) shows the average node coverage for the overlay construction algorithms. When there are no failures, `randk` is the only algorithm that doesn't achieve 100% coverage. The Figure 9.1(b) shows the probability of full coverage for the same algorithms and is interesting to note that `tree+RF` decreases quickly as the percentage of node failures increase.

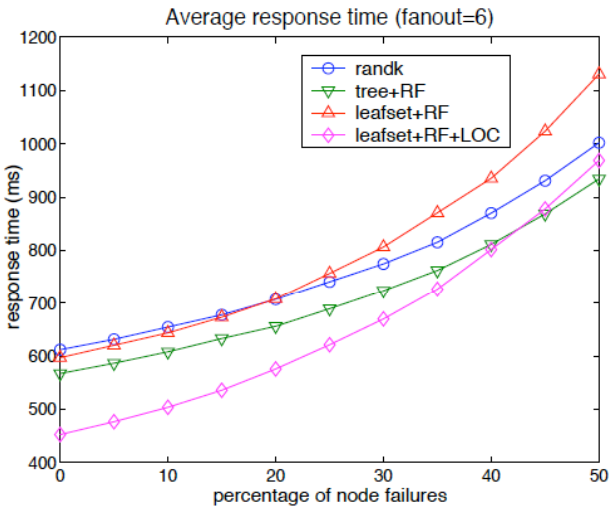
The Figure 9.1(c) shows response time of the algorithms. The `leafset+RF+LOC` has the lowest response time for small node failures; at high failure rates, `leafset+RF` and `leafset+RF+LOC` worst perform because they create deep trees due to the message propagation along the membership ring.

The Figure 9.2 shows the experiment results on PlanetLab, where MON deployed 324 nodes and the overlay tree was built with `k` = 4, 5, 6, 8 for `randk` algorithm and `k` = 4 for the `leafset+RF+LOC` algorithm. Figure 4.12(a) shows the coverage achieved by the two algorithms; `leafset+RF+LOC` covers more than 318 nodes on average, wile `randk` with the same message overhead (i.e., `k` = 4) covers only 305 nodes. `randk` has a lower coverage than `leafset+RF+LOC` even when `k` = 8. The `leafset+RF+LOC` algorithm uses more bandwidth than `randk` because of the leafset addition in the membership view. This bandwidth is very small, considering that each gossip message is sent every 10 seconds and the additional payload is les than 100 bytes. The advantages provided by `leafset+RF+LOC` are well worth this additional overhead.



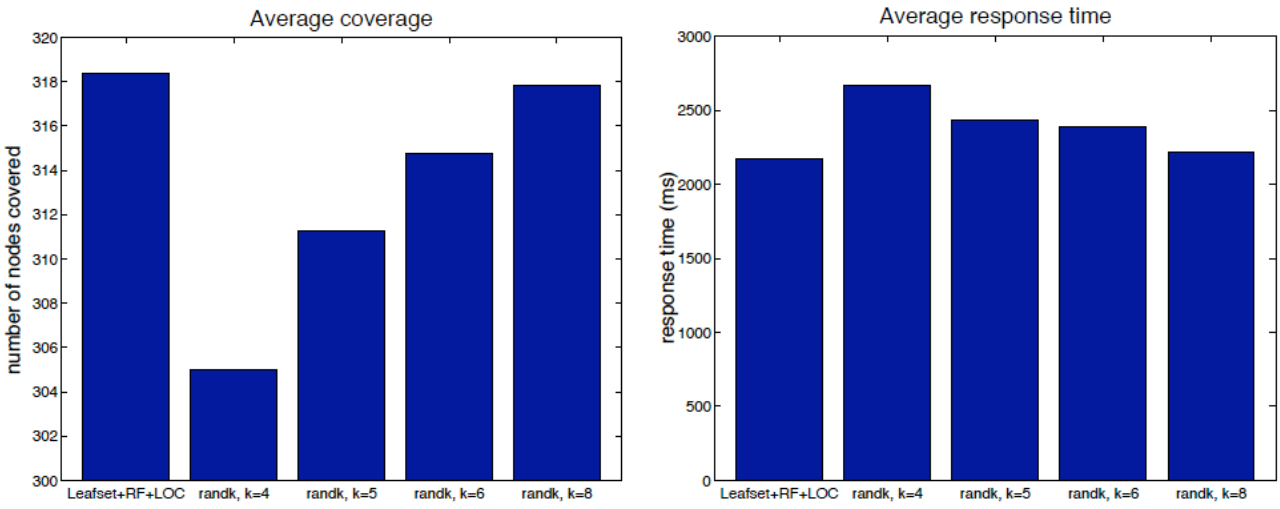
(a) Average node coverage

(b) Probability of full coverage



(c) Response time

Figure 9.1: Comparison of the tree construction algorithms with simulation



(a) Average node coverage (b) Average response time

Figure 9.2: Results of coverage and response time on PlanetLab

MON creates an on-demand overlay and does not recover from failures, hence it was tested the lifespan of an overlay tree before it becomes unusable. An unusable overlay is an overlay where there is a disconnection of `max_drop` nodes from the overlay since it was constructed. Tests were performed on PlanetLab with on-demand trees and DAGs with different number of parents; the value of `max_drop` used in the experiments was 5. The Figure 9.3 shows the results of that test.

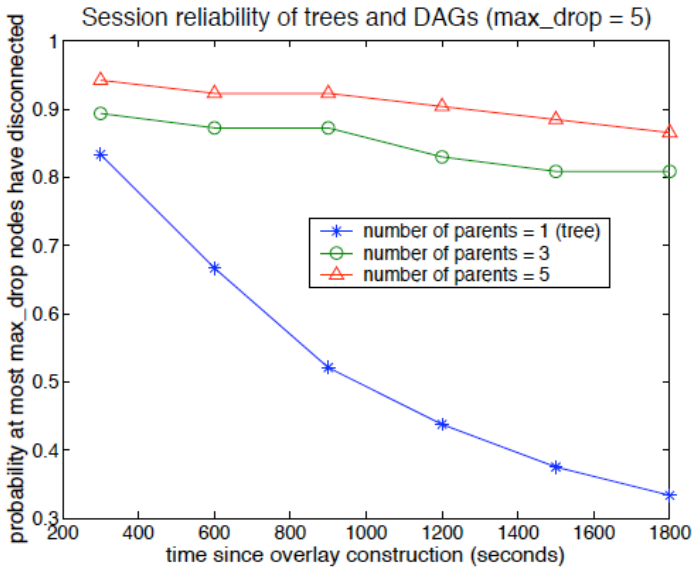


Figure 9.3: Comparison of overlay reliability of tree and DAGs

The results show that a tree overlay has a probability lower than 50% to be usable after 20 minutes, while the probability for the DAG to be usable is more than 80%.

The last experiment involved the analysis of queries of distributed log files (the distributed grep query). A log file with size between 500KB and 5MB was created on each PlanetLab machine, where six of those logs contained the word “Fail”. The query was repeated about 1500 times, and led to the results shown in Figure 9.4.

We can state that all six nodes were found about 10% of the time; at least five 40% of the time, about 90% of times at least four nodes were discovered. The average number of discoveries was 4.36 and the average query execution time was just over 2 seconds.

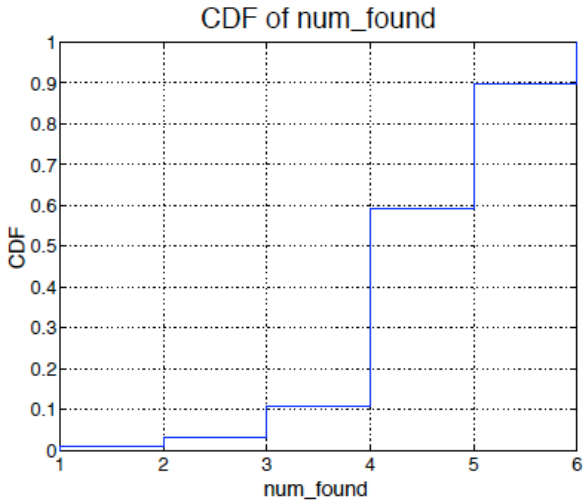


Figure 9.4: Cumulative Distribution Function (CDF) of the nodes found by the query

The reason why MON does not identify all interesting nodes all times motivated by the usage of UDP for communication between nodes, and consequent message loss can influence the final value of nodes found. The retransmission count is 6 for performance reasons, but the developer can specify a different value to have a better tradeoff between query completeness and response time.

9.2. T-MON testbed

This section describes the planned T-MON testbed. Because of issues with the `monclient` application provided with MON code, it was not possible to perform the querying as there would be no opportunities to submit queries to the T-MON system.

For the evaluation, we selected the following metrics:

- query response time, i.e., the time it takes for the query to return a result to the user;
- query coverage, i.e., the number of nodes involved in the query with respect to the total number of nodes in the system;
- threshold time, defined as the 80th percentile of the queries response time.

The testbed included two scenarios: in the first one, called WAN scenario, there would be about one hundred T-MON instances running on PlanetLab nodes, and no more than one T-MON instance would run on each PlanetLab node. In the second one, called LAN scenario, we tried to simulate a TEEVE network allocating T-MON instances on nodes belonging to the same institution.

The number of T-MON simulated instances in either scenario would have a scale of one hundred instances (i.e., the aim was to simulate about one hundred T-MON instances).

We planned to do at least 50 runs of the same query to get strong validation results; we would have tested different typologies of queries. We planned the execution of three queries: one with the MON syntax, to evaluate MON performances in an environment with less nodes than its designed environment. The second query would be a simple T-MON query (i.e., with one `<condition>` and one `<bool_expr>`; as discussed in sections 5.5 and 7.2), while the third query would be a complex T-MON query (i.e., with more than one `<condition>` or `<bool_expr>`), to evaluate how much the complexity of a query would affect the query response time.

9.2.1. WAN scenario

In the WAN scenario, we would simulate about one hundred T-MON instances on nodes of PlanetLab network. We would get the list of PlanetLab nodes where to deploy T-MON from the hostlist discussed in section 6.1.2. We would select the nodes trying to distribute them uniformly across the PlanetLab network (and consequently trying to geographically distribute them across the world). No more than one T-MON instance would run on the same node.

This scenario would simulate the multi-site TEEVE, where the T-MON instance is on each gateway and many locations are running.

The CoMon [COMON] application would allow for monitoring of the PlanetLab nodes executing T-MON, for a better overall point of view about the system performances.

9.2.2. LAN scenario

In the LAN scenario, we would have simulated about one hundred T-MON instances on PlanetLab nodes. The characteristic of this testbed scenario is the selection process of the nodes: we want to cluster nodes together so that some instances of MON are very close to each other but geographically distant from the other instances. The hostlist discussed in section 6.1.2 helped to find Institutions with several PlanetLab nodes, preferably at least four, with the assumption that nodes belonging to the same Institution are connected on a LAN, or at least present a short network distance path. The only exceptions are the PlanetLab hosts under the `6planetlab.edu.cn` domain, that do not share a common geographical location, but are spread across China. The number of the desired Institutions (i.e., Institutions with at least four PlanetLab nodes) is lower than the number of MON instances we would have planned to run, therefore we would use nodes from Institutions with two or three PlanetLab nodes, and we would have executed more than one instance of T-MON in some nodes. This because of the assumption that the CPU and network overhead for T-MON instances are low, therefore the performances of a node would

not be affected by more than one running instance of MON. The T-MON instances, like MON, can run together with other T-MON instances on the same node, as long as they use a different socket port and have a different nodeId.

The CoMon [COMON] application would have supported the monitoring of T-MON instances and the PlanetLab node load for quick troubleshooting.

10. Future work

The Teleimmersion system needs a monitoring application to aid and speed up the development process. The T-MON system may evolve as follows:

- **Completing the T-MON project**. This thesis grouped together all the pieces of information abut MON, and designed an ad-hoc query language for it. The future work may modify the initial design decisions according to the experimental results about the topology and, if possible, through the analysis of real TEEVE trace files.
- **Implement InfoEye into T-MON.** InfoEye is a system that can combine dynamic querying and continuous monitoring to minimize the information management overhead [JL07]. To achieve this, it dynamically moves toward a push model or toward a pull model according to the number of incoming queries, i.e., InfoEye can quickly reconfigure itself to adapt to the changes. Moreover its design, like MON, targets large-scale distributed systems. This would be a great addition to T-MON, allowing reducing the overall resources overhead.
- **Implement multithreading on T-MON**. MON and T-MON are based on the Protocol Plug-In (PPF) framework, whose design implies a single threaded application. We believe that to guarantee the future increase in

offered functionalities, an event-based application such as T-MON should not have a single thread, that will eventually become a bottleneck.

- **Extend the T-MON query language**. There are many TEEVE functionalities that are suitable for monitoring, or modifications to make the querying language more similar to the natural language. One of the latter changes would, for example, be the implementation of Boolean values. A sample query would be the following: `select camera where faulty`. the `faulty` keyword represents a Boolean value and is intended as shortened version of the following: `faulty = true`. The T-MON query language design attempted to use the `value` keyword as workaround, as illustrated in section 7.2.1.

11. Conclusions

This thesis has presented T-MON (Teleimmersion-MON), a distributed monitoring application for Teleimmersion system, also referred to as TEEVE (Tele-immersive Environment for EVErybody). We described ad analyzed the TEEVE system to understand its main features and its critical resource usage, to better design a system with the smallest possible impact on performances.

We believe that Teleimmersion is a promising technology, that one day it will be a breakthrough in the personal communication, changing the lifestyle of many people. People will have to travel less, and they will be able to have more meetings with people all around the world, without the constraint of the location (currently when traveling there is the possibility to only attend meetings within a limited distance from the location). Telemedicine may benefit, providing qualified support to people living in rural and remote areas, supporting virtual visits as well as virtual remote care. The 3D cameras get smaller and better, currently there are implementation of some prototypes of 3D cameras as big as conventional webcams. The applications are moving towards a distributed paradigm, as well as operating systems, providing a future legacy infrastructure for the development of distributed applications like TEEVE and MON.

The development of this thesis provided learning on many subjects, both technical and human. It provided the opportunity to learn new skills, such as the Python and Perl programming languages, greatly improved my shell scripting capabilities (overcoming its many limits, such as the management of filenames containing white spaces) and, most importantly, have a broader view of the leading edge research, thanks to the attendance to several talks and Distinguished Lectures. It improved the planning capability, to deal with deadlines efficiently and improved the capability to define an accurate schedule. But, aside the technical skills, the greatest learning was on the human side: as the project presented several technical hurdles

and its progress was slow, there was the fear to not make others feel the passion dedicated to the project, as the judgment comes from the results achieved. Too many problems may also lead to think that they are excuses to justify little work on the project. The answer to that was stronger self-motivation and harder work.

Future work could complete the T-MON project, considering that we documented many of the previously undocumented MON features, including the code execution flow, the complete grammar of the MON query language, and a step-by-step guide to add a new language keyword to MON.

Another interesting idea for a future work may be the implementation of the InfoEye query pattern in MON, adding to MON the capability to dynamically switch from a push-based to a pull-based query model. The single threaded model that T-MON and MON use may soon become a bottleneck as the application evolves with the support for more functionalities, therefore future work may focus on the redesign of the Protocol Plug-in Framework (PPF), on which T-MON and MON are based, to support multi-threaded applications.

Finally, another interesting development is the extension of the query language to support the monitoring of more TEEVE parameters or to make its syntax closer to the natural language; as an example, it may implement Boolean operands, to support queries like the following: `select camera where faulty`.

12. References

[CLR+05] Yatin Chawathe, Anthony LaMarca, Sriram Ramabhadhran, Sylvia Ratnasamy, Joseph Hellerstein, and Scott Shenker, *A case study in building layered DHT applications*, Tech. Rep. IRS-TR-05-001, Intel Research, January 2005.

[COMON] CoMon Homepage. http://comon.cs.princeton.edu/.

[EXIT] `exit` man page. http://pwet.fr/man/linux/commandes/posix/exit .

[GS03] Jun Gao, Peter Steenkiste, *Efficient Support for Range Queries in DHT-based Peer-to-Peer Systems*, Technical Report CMU-CS-03-215, Carnegie Mellon University, December 2003.

[GS07] Jun Gao, Peter Steenkiste, *Efficient Support for Similarity Searches in DHT-based Peer-to-Peer Systems*, in IEEE International Conference on Communications (ICC '07), 2007.

[GSM07] Alejandra Gonzalez Beltran, Paul Sage and Peter Milligan, *Skip Tree Graph: a Distributed and Balanced Search Tree for Peer-to-Peer Networks*, IEEE International Conference on Communications, 2007. ICC '07.

[INT96] Internet2 Homepage. http://www.internet2.edu/ .

[JL07] Jin Liang, *Building and Managing Large Scale Distributed Services*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October, 2007.

[JM+03] Mark Jelasity, Alberto Montresor, and Ozalp Babaoglu, *Gossip-based aggregation in large dynamic networks*, ACM Transactions on Computer Systems, August 2003.

[KYG+08] Steve Ko, Praveen Yalagandula, Indranil Gupta, Vanish Talwar, Dejan Milojicic, Subu Iyer. *Moara: Flexible and Scalable Group-Based Querying System*, Proc. ACM/IFIP/USENIX Middleware, 2008.

[LYY+05] Jin Liang, Zhenyu Yang, Bin Yu, Yi Cui, Klara Nahrstedt, Sang-Hack Jung, Art Yeap and Ruzena Bajcsy, *Experience with Multi-Camera Tele-Immersive Environment*, in Proc. of NSF Experience Workshops on Pervasive Computing and Cluster Computing (invited), 2005.

[LN05] Jin Liang and Klara Nahrstedt, *RandPeer: Membership Management for QoS Sensitive Peer-to-Peer Applications*, in Proc. of IEEE Infocom 2006, April, 2006.

[MON05] MON Project Homepage. http://cairo.cs.uiuc.edu/projects/mon/ .

[MONCGI] MON web interface. http://cairo.cs.uiuc.edu/cgi-bin/webmon/webmon.cgi .

[MONET] MONET Research Group Homepage. http://cairo.cs.uiuc.edu/ (Multimedia Operating Systems and Networking).

[MYSQL] MySQL Homepage. http://www.mysql.com/ .

[OSSH02] OpenSSH Homepage. http://www.openssh.com/ .

[PDN023] PlanetLab Implementation Team, *PlanetLab: Version 3.0, PDN-04-023*, PlanetLab Design Notes (PDN), August 2004 (updated January 2005). http://www.planet-lab.org/files/pdn/PDN-04-023/pdn-04-023.pdf .

[PL02] PlanetLab Homepage. https://www.planet-lab.org/ .

[PLDEV] Marc E. Fiuczynski, *migrating to centos*, Planetlab-devel mailing list, Dec 15, 2008. http://lists.planet-lab.org/pipermail/devel/2008-December/003173.html .

[PSQL] PostgreSQL Homepage. http://www.postgresql.org/ .

[RD01] Antony Rowstron and Peter Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, in Middleware 2001, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker, *A Scalable Content-Addressable Network*, in Proceedings of ACM SIGCOMM'01, 2001.

[RRH+04] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph Hellerstein, and Scott Shenker, *Brief announcement: Prefix hash tree*, in ACM PODC'04, July 2004.

[RSYNC] Rsync Homepage. http://rsync.samba.org/ .

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, in Proceedings of ACM SIGCOMM'01, 2001.

[SMZ03] K. Sripanidkulchai, B. M. Maggs, and H. Zhang, *Efficient content location using interest-based locality in peer-to-peer systems*, in IEEE Proceedings of the INFOCOM, 2003.

[SSH] SSH (Secure SHell) protocol. http://en.wikipedia.org/wiki/Ssh .

[TSG07] Computer Science Subversion Service web page. https://agora.cs.illinois.edu/display/tsg/Subversion+Service , Technology Service Group (TSG) at the Department of Computer Science at the University of Illinois at Urbana-Champaign.

[VX04] Vxargs Homepage. http://vxargs.sourceforge.net/ .

[WPCHO] Chomsky Hierarchy, English Wikipedia. http://en.wikipedia.org/wiki/Chomsky_hierarchy .

[WYN+07] Wanmin Wu, Zhenyu Yang, Klara Nahrstedt, Gregorij Kurillo, and Ruzena Bajcsy, *Towards Multi-site Collaboration in Tele-immersive Environments*, in Proc. of 15th annual ACM international conference on Multimedia (MM'07) (short paper), 2007.

[YCL+05] Zhenyu Yang, Yi Cui, Bin Yu, Jin Liang, Klara Nahrstedt, Sang-Hack Jung and Ruzena Bajscy, *TEEVE: The Next Generation Architecture for Tele-Immersive Environments*, in Proc. of the 7th IEEE International Symposium on Multimedia (ISM'05), Irvine, CA, 2005.

[YYD+06] Zhenyu Yang, Bin Yu, Ross Diankov, Wanmin Wu and Ruzena Bajcsy, *Collaborative Dancing in Tele-immersive Environment*, in Proc. of ACM Multimedia (MM'06), Santa Barbara, CA, 2006.

[YYW+06] Zhenyu Yang, Bin Yu, Ross Diankov, Wanmin Wu, Klara Nahrstedt, Ruzena Bajscy, *A Study of Collaborative Dancing in Tele-immersive Environments*, in Proc. of The 8th IEEE International Symposium on Multimedia (ISM'06), 2006

[ZSY+07] Yaodong Zhang, Guobin Shen, Yong Yu, *LiPS: Efficient P2P Search Scheme with Novel Link Prediction Techniques*, in IEEE International Conference on Communications (ICC '07), 2007.

[ZY08] Zhenyu Yang, *Multi-stream management for supporting multi-party 3D tele-immersive environments*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2008.