Proceedings of the International Conference on Communication and Computer Networks
Marina del Rey, California, pp. 202-207, October 2005.

Efficient Client-Transparent Fault Tolerance for Video Conferencing

Navid Aghdaie and Yuval Tamir
Concurrent Systems Laboratory
UCLA Computer Science Department
Los Angeles, California 90095
{navid,tamir}@cs.ucla.edu

ABSTRACT

As video conferencing plays an increasingly critical role in many business environments, there is a need to ensure highly reliable operation of the conferencing infrastructure. We present a scheme for adding fault tolerance to an existing video conferencing server. The scheme is client-transparent so that it can be used by the installed base of clients. While the scheme is based on replication, the associated overhead is negligible since the backup does not process the media streams. Most previous work on fault-tolerant network services focused on transaction-oriented services. Video conferencing is an interesting test-case for applying fault tolerance for other types of services since it combines critical conference state that must be protected with media streams where limited data losses are acceptable. Our implementation combines kernel modules with small changes to the server application to efficiently preserve both the reliable connections used for control messages and unreliable connections used for media transfer.

KEY WORDS
video conferencing, reliable network services, replication

1. Introduction

High reliability and availability are key requirements for critical network services such as online banking. Less critical network services, such as instant messaging and video conferencing, are increasingly being relied on by consumers who expect trouble-free operation. Delivery of high availability and reliability for a variety of network services will thus translate directly into increased customer satisfaction and profits. One of the main causes of service disruptions is server faults. Hence, fault tolerance techniques that allow providers to deliver continuous uninterrupted service despite server faults will become increasingly important.

For many services (web service, video conferencing) there is a large installed base of clients and it is not practical to require all of those to be modified. Hence the fault tolerance scheme used should be client-transparent, i.e., operate without requiring any special action by the client. Most of the existing work on client-transparent fault-tolerant network services has focused on web service [1, 3, 8, 10, 13], dealing with TCP connections, HTTP transactions, and web servers. This paper demonstrates how the methodology and mechanisms developed in the context of web service can be adapted to a very different type of service — video conferencing.

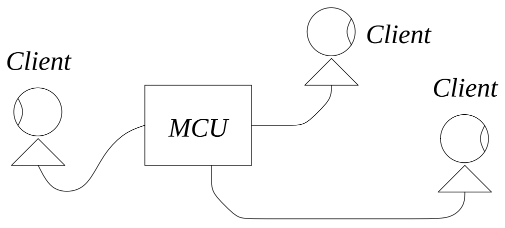

Figure 1: Video Conferencing with a Multi-Conferencing Unit (MCU). An MCU fault causes the conference to fail.

Off-the-shelf video conferencing schemes consist of multiple clients and a Multi-Conferencing Unit (MCU) that is the conference central server (Figure 1). The MCU executes the conferencing application and maintains the state of the conference. Clients connect to the MCU to join a conference. The main function of the MCU is to route the multimedia stream (i.e. video) from each client to others. The MCU application may also provide additional functionality such as the mixing of multiple media streams, echo cancellation, content-adaptation to support different types of clients, and conference management such as maintaining multiple virtual rooms.

Since conference state is kept only at the MCU, an MCU failure results in the failure of all active conferences handled by that MCU. The goal of our work is to ensure the survival of the MCU service despite the failure of an MCU process or host. The MCU state changes during the conference as new clients join, some clients depart, etc. These changes must be reliably preserved across faults. Fortunately, most of the MCU processing requirement is due to the processing of the media streams, where absolute reliability is not required to maintain the conferencing service. Our approach to adding fault tolerance to the MCU judiciously combines fully replicated operation for maintaining critical conference state with a very low overhead failover mechanism for handling the media streams themselves.

The key contribution of this paper is the presentation and evaluation of an efficient fault tolerance scheme for video conferencing. This scheme is an adaptation of the fault tolerance methodology that we developed previously for web service [1, 2] and serves as a demonstration that the methodology can be extended to other network services. To our knowledge, this paper would be the first publication of a practical client-transparent fault tolerance scheme for video conferencing. Section 2 discusses the details specific to the design and implementation of our fault-tolerant video conferencing scheme using OpenMCU [11], an open source H.323 [7] conferencing server developed on top of the OpenH323 library [11]. Section 3 presents the performance evaluation results. Related work is discussed in Section 4.

2. Fault Tolerance for a Conferencing Server

Our video conferencing implementation is based on OpenMCU [11], an open source H.323 [7] conferencing server developed on top of the OpenH323 library [11]. The OpenMCU application maintains virtual conference rooms, receives and mixes media from the clients in a conference, and periodically transmits the mixed media to each client. TCP connections are used as the reliable channel for transmission of control messages. The media is transferred on top of unreliable channels using the UDP protocol.

The methodology for adding fault tolerance to the video conferencing service is based on our scheme on fault-tolerant web service [1, 2], called *CoRAL*. To achieve client-transparent fault tolerance the service identity, communication connection state, and relevant application state must be preserved across faults [2]. We preserve the required state on a replica on the same LAN subnet. During fault-free (duplex) operation, the MCU connection state is actively replicated and the application state is synchronized as necessary in order to handle non-deterministic behavior. If one of the replicas fails, the surviving MCU takes over and operates in simplex mode.

Our implementation requires changes to the MCU at OS and application levels. A kernel module (roughly 5000 lines of code) on each replica facilitates the functionality necessary for replicating the connection state. The kernel-level code is application independent. The TCP portion is exactly the code that was used for CoRAL [1, 2], and the UDP code developed for this work can be used with other services that use UDP connections. Modifications to the MCU application code (5000 lines) implement application level synchronization and fault detection. Some (1000 lines) of our user-level code, whose main functionality is the synchronization of application-level state, is application specific and was written specifically for the OpenMCU application. The rest of the code, which mainly deals with fault-detection and failover, is mostly application independent and similar to CoRAL. The implementation details are presented in the rest of this section.

2.1. Identity Preservation

In duplex operation, one of the MCU hosts is configured with the service identity, i.e., the advertised IP address. If this host fails the surviving host must take over the advertised IP address in order to achieve client transparency. When a fault is detected, the surviving MCU establishes an additional IP address alias (the advertised address) using a Linux *ioctl* call and uses gratuitous ARP to inform LAN's router of this takeover. As a result, any client packets sent to the advertised service address are routed to the surviving MCU [2].

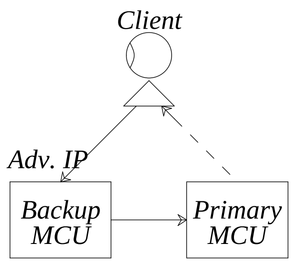

Figure 2: TCP connection replication. Incoming TCP packets arrive at the backup and are forwarded to the primary. Only the primary sends outgoing acknowledgment in fault-free operation. Hence, client will not receive an acknowledgment unless both replicas have received a copy its packet.

2.2. Reliable Communication

The MCU control channels require reliable communication and thus operate on top of TCP. We used part of CoRAL [1, 2] to actively replicate the state of TCP connections to the MCU. CoRAL transparently multicasts client TCP packets to two replicas and ensures that the TCP stack state and acknowledged client packets can be recovered if one of the replicas fails. In fault-free operation, the advertised address for the conference is mapped to the backup MCU. Hence, incoming client packets arrive at the backup first. Upon the arrival of an incoming client TCP packet at the backup, a copy of the packet is forwarded to the primary MCU (Figure 2). Only the generated acknowledgment by the primary is sent to the client. The acknowledgment generated by the backup's TCP stack is discarded. As a result, the client will not receive an acknowledgment unless both the primary and backup have received a copy of the client packet. Hence, if a fault occurs, the backup can seamlessly take over the handling of active TCP connections. Further details of TCP connection failover are described in CoRAL papers [1, 2].

2.3. Unreliable Communication

The MCU media channel is built on top of UDP. In order to preserve the media channel across faults, identical UDP socket structures must be created on both MCU replicas. In addition, client UDP packets sent to the faulty MCU must be transparently routed to the surviving replica after a fault. H.323 services typically create the media stream after receiving a client request on the control channel. With OpenMCU, UDP sockets are created in response to client requests received on the TCP connection. Our scheme multicasts requests on TCP connections to both replicas. Hence, if identical applications are executed on both replicas and the application behavior is deterministic, or non-determinism is handled as shown later in this section, then identical UDP socket structures are created on both replicas.

The key to the low overhead of our scheme is that the UDP packets (the media streams) are not really processed by the backup. The handling of UDP packets by replicas can be implemented using several configurations. We considered and implemented three different approaches. One approach is the use of forwarding similar to our TCP replication (Figure 2). In fault-free operation, the service IP address for the media stream is mapped to the backup. As a result, client packets arrive at the backup first. Upon the receipt of client UDP packets, the backup forwards a copy to the primary by changing the destination IP address in the packet. Outgoing packets are transmitted only by the primary, with the source IP address always being set to the advertised address to maintain transparency. Faults may occur on either primary or backup replicas. If the primary MCU fails, the backup no longer forwards packets to the primary, and it starts processing incoming UDP packets and transmitting outgoing packets to clients. If the backup MCU fails, the primary takes over the identity of backup server and client packets arrive directly at the primary.

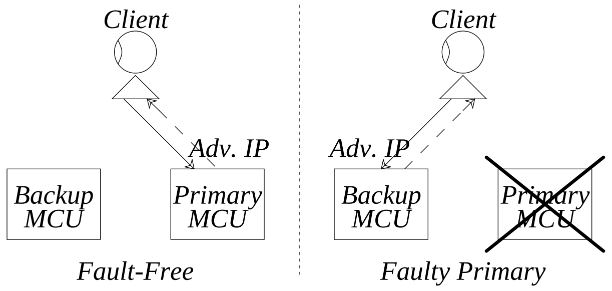

Figure 3: Direct UDP communication between client and primary. Advertised IP address is mapped to the primary in fault-free operation. If primary MCU fails, the backup takes over the advertised address and continues operation in simplex mode.

Another approach is direct UDP communication between clients and the primary (Figure 3). During fault-free operation, the service IP address for the media stream is mapped to the primary and UDP packets are exchanged directly between the client and primary MCU. If the primary MCU fails, the backup takes over the service IP address, receives incoming client packets, and transmits outgoing packets to clients. If the backup MCU fails, the communication between the clients and primary can continue normally, albeit now without fault-tolerance features. The direct client to primary communication eliminates the forwarding overhead that is incurred with the forwarding approach. However, since our TCP replication implementation maps the service address to the backup in fault-free operation, this approach requires the decoupling of reliable and unreliable connections at the application protocol level. The H.323 protocol includes this features. The advertised address of the unreliable channel is given to the client by the server over the control channel. Hence, the service IP address for the control and media channels need not be the same. However, this feature is not commonly used. In fact the openh323 library implementation assumes that the two addresses will always be identical. We made a small modification to the library, allowing the use of different IP addresses for the TCP and UDP connections.

A third approach is to use IP multicast. An IP multicast address is used as the advertised address of the media stream. Hence, the network multicasts the client UDP packets to both replicas (Figure 4). In fault-free operation, only the primary transmits outgoing packets to

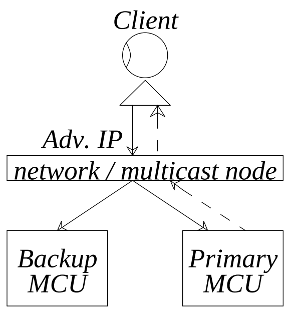

Figure 4: UDP communication using IP multicast or multicast node. Client UDP packets are multicast to both replicas by either the network (IP multicast) or a multicast node. The advertised address is an IP multicast address or address of multicast node, respectively. In fault-free operation only the primary replica sends outgoing packets to the client.

the client. If the primary fails, the backup takes over. If the backup fails, no changes take place at the primary. The use of global IP multicast addresses for such small scale multicast may not be practical. As an alternative, a server-side node or router can be used to multicast client UDP packets and achieve the same effect. The advertised IP address for the media is mapped to the multicast node. The multicast node simply sends a copy of each client packet that it receives to both replicas. A drawback of such a local multicast node is that it becomes a new single point of failure. However, in practice, since the multicast node is stateless and performs a very simple operation, it is likely to be highly reliable and, if necessary, it would be relatively simple to make this multicast function fault-tolerant. All that is required is a backup that detects faults and simply takes over the IP address and continues the processing of packets in the same manner. The performance comparison of the three approaches is presented in Section 3.

2.4. Application State and Non-determinism

The MCU application can be actively replicated, with identical copies available and running on both replicas. Since the application input, i.e., TCP and UDP connections, are replicated, the application state will also be identical if the processing is deterministic. However, the OpenMCU application is not deterministic. Hence, the applications on the primary and backup must be synchronized whenever there is a non-deterministic state change. We found only a few such events: the selection of initial sequence number and SyncSourceOut variable used by the RTP (media) connection, and creation of the UDP port used for media transfer. Fortunately, these events all occur at the creation and initialization of each RTP session and therefore can be synchronized together. We synchronize the replicas by exchanging messages. The primary sends the non-deterministic state changes to the backup. The backup makes state changes according to the primary's message — it sets initial values for variables and creates UDP ports. The primary continues after receiving an acknowledgement from the backup. Hence, the application states will be identical before the RTP session is used. If the primary fails during the synchronization procedure before the synchronization message reaches the backup, the primary state changes are never visible to the clients and the backup can safely make its own non-deterministic state changes.

If external processes are allowed to use resources on the replica hosts, synchronization of non-deterministic state changes as described above may not be possible. For example, the backup may not be able to create a UDP socket with the same port number as the primary because another process is using it. To get around such resource conflicts, we added the possibility of a negative acknowledgment response to the synchronization messages. If the backup cannot make the required state changes, it sends a NACK message back to the primary, informing it to undo its state changes and try a different path. In the UDP example, the primary deallocates the socket structure, creates a new socket bound to a different port number, and retries synchronization with the backup.

Active replication of MCU application inherently incurs a large performance overhead. The backup application performs every operation performed by the primary. Some application operations may not affect the application state and are not required to be performed by a replica if the only goal is to preserve the application state. Specifically, there is no critical state at the MCU associated with media processing. During fault-free operation, it is not necessary for the backup MCU to process or generate media. Only the control stream which affects the application state must be processed. Hence, as an optimization, we modified OpenMCU and disabled all operations related to the media stream in backup mode while keeping the control operations intact. Our evaluation results in section 3 show that we successfully eliminated almost all the processing on the backup.

2.5. Fault Detection and Fail-Stop Assumption

We assume that only a single MCU host at a time may fail. We further assume that server processes are fail-stop [12]. Hence, faults are detected using heartbeats and timeouts. A heartbeat generator and a heartbeat monitor process were added to the OpenMCU application. Each generator periodically transmits a sequenced UDP packet to the other MCU. The monitor receives these packets, and determines that a fault has occurred when there are consecutive missed heartbeats.

In practice, faults are not always fail-stop. Faults may be Byzantine, resulting in arbitrary behavior, such as transmission of unwanted messages from faulty nodes. Unfortunately, Byzantine faults cannot be handled transparently for TCP connections. A faulty node (e.g., MCU) may send TCP reset or FIN packets for the connection, causing the remote end (e.g., client) to abandon or close the connection. Once the client-side connection state is lost, the TCP connection cannot be transparently restored. Hence, client-aware solutions are required in order to handle Byzantine faults.

Although we cannot handle Byzantine faults, we can handle both host and process fail-stop faults. A key problem is that when a user process crashes while the host kernel is still running, reset or FIN TCP packets may be sent to the client, causing the TCP connection to be terminated. To handle this problem, our kernel module discards reset or FIN TCP packets when a process crashes. Specifically, when a process crashes, the kernel implicitly calls exit and closes all associated sockets. Our kernel module detects any implicit exit calls for MCU processes and discards any TCP FIN or reset packets generated due to the implicit exit close calls. This ensures that the client connection is not destroyed and a seamless failover to the backup can occur. Our experiments show that MCU application processing is mostly (94 percent) at user-level. Fault injection experiments on other systems have shown that if almost all the time is spent in application user mode, most of the faults either have no effect or cause the process to crash [9]. Hence, the ability to maintain connections and correctly failover when a process crashes but the host continues to run, is critical to the reliability of our scheme.

3. Performance Evaluation

Evaluation experiments were performed on 2.6 GHz Intel Pentium IV Xeon PC's interconnected by a gigabit/second switched network. The MCU nodes were running our modified OpenMCU application on top of the Linux 2.4.20 kernel with our kernel module installed. We used three client nodes, each running the ohphone [11] application with a Logitech Quickcam Pro 3000 webcam. The webcams were pointed towards monitors where the screen refresh (i.e., flicker) created a constantly changing image. Finally, the three clients joined a conference hosted on the MCU.

We emulated faults by physically disconnecting an MCU from the network (host crash), killing the MCU application processes with the kill command (process crash), and by randomly flipping bits in the processor's registers using a kernel-based fault injector. Our implementation successfully recovered from all types of crash faults. 25.4 percent of fault injector injections caused a crash (which were recovered), with the rest not having a noticeable effect on normal operation.

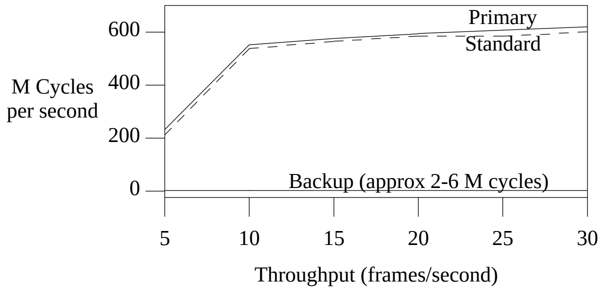

Figure 5: CPU cycles (million per second) used by a standard off-the-shelf MCU and the primary and backup MCU's in our scheme. System throughput was varied configuring clients and MCU to use video with different number of frames per second.

Qualitatively, faults cause a brief interruption of moving video at each client, with the duration of the interruption being on the order of the heartbeat period (sub second). If the video is static and not moving, there is a noticeable (few seconds) impact on the image sent to the clients. The reason is that the replica taking over (i.e.

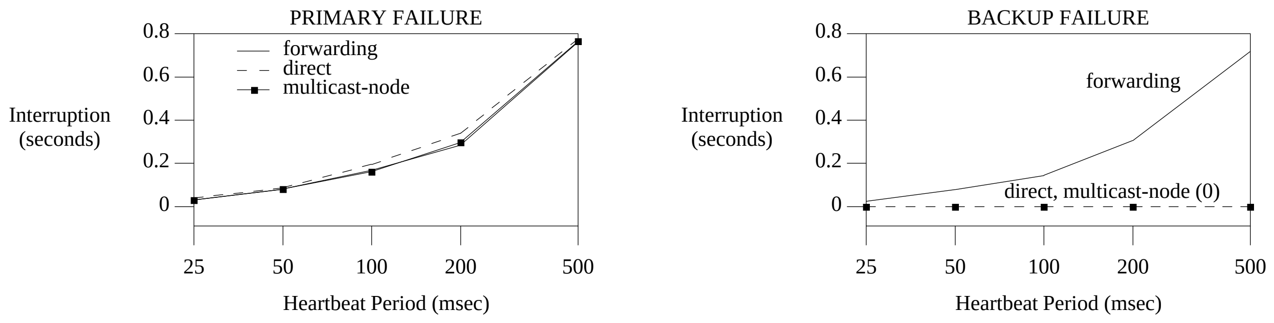

Figure 6: Interruption time due to failover for different UDP schemes. Primary (left) and Backup (right) MCU faults are shown.

backup) starts with a blank image. The video image is partitioned into multiple chunks and the client codec transmits the static parts of the video image less frequently. Hence, a few seconds are required for the backup to receive the entire image. Fortunately, this has essentially no impact on the ability to continue the conference since audio does not suffer from this problem. Specifically, the audio codec does not rely on history. Hence, the audio interruption duration is always on the order of the heartbeat frequency, and is barely noticeable with typical (<100msec) heartbeat settings.

3.1. Processing Overhead

We measured the processing cycles used by the MCU using the CPU's performance monitoring registers. We measured global_power_events [6] which accumulates the time during which the processor is not stopped. CPU cycles used by a bare system were deducted to derive the actual cycles used by the MCU. The heartbeat period was set to 100 milliseconds. The system throughput was varied by configuring the clients and MCU to transmit video at different number of frames per second.

Figure 5 show the CPU cycles used by our primary and backup MCU and a standard MCU without fault-tolerance features. The backup MCU is mostly idle. It uses two to six million cycles per second depending on the UDP connection handling (discussed below). The primary consumes slightly more cycles than a standard server mainly due to bookkeeping operations in the kernel module and heartbeat generation and monitoring. The overall processing overhead is the difference between the primary and standard cycles plus the cycles used by the backup. This overhead is small — roughly 3 percent for the direct UDP scheme at 30 frames per second.

3.2. Failover Latency

When a MCU fails, the system is briefly unavailable while a failover takes place. The failover time includes the time to detect a fault (i.e., consecutive missed heartbeats) and the time for the reconfiguration of system from duplex to simplex mode. During failover, client packets sent to the MCU may be lost, potentially resulting in a noticeable interruption to the clients. We quantified the interruption duration by using tcpdump to monitor packets to the clients and recording the ‘‘gap’’ in packet flow. In this experiment faults were injected into the system by disconnecting one of the MCU replicas from the network.

Figure 6 shows the interruption time due to failover caused by primary or backup MCU faults. The heartbeat rate has a major effect on the length of interruption, showing that failover time is dominated by the fault detection time. Primary faults cause an interruption in all schemes. The forwarding and multicast-node schemes' failover times are virtually identical. The direct scheme has a slightly higher failover time because the backup must takeover the primary's IP address which is used as the identity of the UDP connection. The other two schemes do not require an address takeover if the primary fails.

Backup faults cause an interruption only with the forwarding scheme. If the backup fails with the forwarding scheme, client UDP packets will not reach the primary until the fault is detected and a failover occurs. Backup faults in the direct and multicast-node schemes do not cause an interruption because client UDP packets are not lost. The primary MCU detects the fault and configures itself for simplex operation without interrupting the media stream. The reconfiguration from primary to simplex is necessary only for TCP connection failover and fault detection processes.

3.3. Impact of Heartbeats and UDP Configuration

Figure 7 shows the percentage of available processing cycles used by the backup MCU with a constant system throughput of 30 frames per second and varying heartbeat rates. Overall, the backup MCU is mostly idle — utilization of less than 1%. The multicast-node scheme has more processing overhead than the direct scheme because the backup node receives, examines, and discards a copy of every client UDP packet. The forwarding scheme has the most overhead because every client packet is copied, it's header rewritten, and then forwarded to the primary. The heartbeat rate affects the processing overhead with all the schemes. Since the heartbeat rate also effects the failover time, there is a tradeoff between processing overhead and failover time.

3.4. TCP and Application Synchronization Overheads

Most of the MCU processing requirement is due to the processing of the media stream. Hence, the evaluation results above are focused on the media stream and UDP connections. However, there is also overhead due to TCP connection replication and application synchronization.

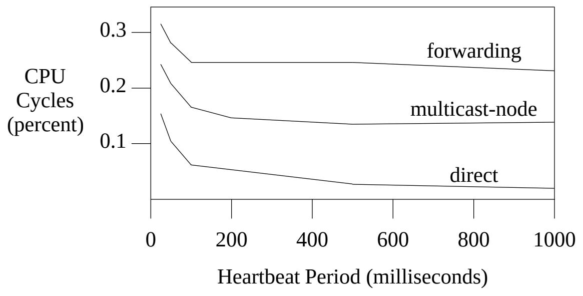

Figure 7: Percent of available CPU cycles used by the backup MCU in duplex mode for different UDP configurations. Throughput was kept constant at 30 frames per second.

To join a conference, a client establishes a TCP connection and sends a control message. With our scheme, the TCP connection is replicated and relevant application state is synchronized between the replicas at join time. We measured the CPU cycles used for the establishment of the control channel and the addition of a client to a conference. Each replica consumes roughly 155 million CPU cycles for the connection establishment and application synchronization. This initialization cost is negligible compared to the millions of cycles used *every* second for the media processing. Other control messages (e.g., connection close, change in throughput) may also be exchanged between the MCU and clients, but they are typically infrequent.

4. Related Work

Much of the early work on fault-tolerant network services was focused on increasing availability and did not recover active connections at failure time [4]. Most client-transparent fault tolerance schemes for network services focus on the Web and the recovery of TCP connections using either replication [1, 8, 10, 13] or logging [3]. Multimedia service reliability research has been mainly focused on broadcast applications based on one way delivery of video from one server to multiple clients. Zagorodnov et al [14] adapted FT-TCP [3] to increase the reliability of Apple Darwin streaming server. To handle application non-determinism, relevant system calls were synchronized to ensure an identical server state would be available despite a fault. End system multicast [5] uses multicast at application level and provides overlay spanning trees for data delivery. Changes to clients or deployment of proxies at strategic locations near the clients are required. Unlike these solutions, our work is based on interactive communication between multiple clients using a stateful centralized MCU, and transparent recovery from faults that may occur at the MCU.

5. Conclusion

We have presented the design and implementation of a client-transparent fault-tolerant video conferencing service that can recover from host or process crashes at the server site. We believe that this would be the first publication of a practical client-transparent fault tolerance scheme for video conferencing. Our implementation is based on an existing conferencing server (MCU) and required modification of a tiny fraction of the existing code. Most of the user-level and kernel-level code in our implementation is directly usable or easily adaptable for other network services. This is verified by the fact that the TCP components were minor adaptations of our previous work on fault-tolerant web service. Our measurements show that the processing overhead of our scheme during fault-free operation is insignificant — approximately 3% additional CPU cycles. Our experiments demonstrated successful recovery from all host crashes and process crashes, including process crashes caused by random fault injection to CPU registers. With a low-overhead configuration (heartbeat period of 25ms-50ms), when a fault occurs, the conference proceeds with no loss of audio (a barely noticeable ‘‘blip’’) and a minor disruption of static video images that clears up in a few seconds.

References

[1] N. Aghdaie and Y. Tamir, ‘‘Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support,’’ *Proceedings of the The 11th International Conference on Computer Communications and Networks*, Miami, Florida, pp. 63-68 (October 2002).

[2] N. Aghdaie and Y. Tamir, ‘‘Fast Transparent Failover for Reliable Web Service,’’ *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 757-762 (November 2003).

[3] L. Alvisi et al., ‘‘Wrapping Server-Side TCP to Mask Connection Failures,’’ *Proceedings of IEEE INFOCOM*, Anchorage, Alaska, pp. 329-337 (April 2001).

[4] T. Brisco, ‘‘DNS Support for Load Balancing,’’ RFC 1794, IETF (April 1995).

[5] Y.-h. Chu et al., ‘‘Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture,’’ *ACM SIGCOMM*, San Diego, CA, pp. 55-67 (August 2001).

[6] Intel Inc, *IA-32 Intel Architecture Software Developer’s Manual - Volume 3: System Programming Guide*, 2004.

[7] International Telecommunication Union, ‘‘Packet-based multimedia communications systems,’’ *ITU-T, Recommendation H.323 V5*, Geneva, Switzerland (May 2003).

[8] R. Koch et al., ‘‘Transparent TCP Connection Failover,’’ *International Conference on Dependable Systems and Networks*, San Francisco, California, pp. 383-392 (June 2003).

[9] H. Madeira et al., ‘‘Experimental Evaluation of a COTS System for Space Applications,’’ *International Conference on Dependable Systems and Networks (DSN’02)*, Washington, D.C., pp. 325-330 (June 2002).

[10] M. Marwah et al., ‘‘TCP Server Fault Tolerance Using Connection Migration to a Backup Server,’’ *International Conference on Dependable Systems and Networks*, San Francisco, California, pp. 373-382 (June 2003).

[11] Quicknet Technologies Inc, ‘‘OpenH323 Project,’’ *http://www.openh323.org*.

[12] F. B. Schneider, ‘‘Byzantine Generals in Action: Implementing Fail-Stop Processors,’’ *ACM Transactions on Computer Systems* **2**(2), pp. 145-154 (May 1984).

[13] G. Shenoy et al., ‘‘HydraNet-FT: Network Support for Dependable Services,’’ *International Conference on Distributed Computing Systems*, Taipei, Taiwan, pp. 699-706 (April 2000).

[14] D. Zagorodnov et al., ‘‘Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP,’’ *International Conference on Dependable Systems and Networks*, San Francisco, California, pp. 393 - 402 (June 2003).