

Increasing the Maximum Call Density of OpenH323 and OPAL

Craig Southeren, Post Increment
craigs@postincrement.com
28 November 2005

The architecture of the OpenH323 and OPAL protocol libraries and the underlying PWLib hardware abstraction interface was fashioned before the advent of readily available, low-cost and high performance computing systems. This architecture has limitations that make it difficult to achieve effective utilization of contemporary hardware platforms. Specifically, the the “one socket, one thread” model causes PWLib processes to reach operating system limits per-process thread counts long before CPU or memory resources are exhausted. This white paper describes the advantages and disadvantages of the current architecture and describes code modifications to solve it.

1. Introduction

Purpose

This document describes and analyses the current architecture of the OpenH323 and PWLib call handling system with emphasis on issues that prevent high call densities within a single program. It also describes modifications to the architecture intended to achieve higher call densities. The intent is to provide a basis for a discussion about future changes to software to remedy the perceived issues.

Scope

This document describes the current call handling architecture and several proposals for new architectures. It assumes a basic knowledge of OpenH323, OPAL, PWLib and the issues regarding threading, scheduling and C++ programming.

References

None

History

Date	Version	Author	Description
11 Nov 05	0.1	Craig Southeren <craigs@postincrement.com>	Initial version
18 Nov 05	0.2	Craig Southeren <craigs@postincrement.com>	Created pre-release version for review
28 Nov 05	1.0	Craig Southeren <craigs@postincrement.com>	First release version

2. Overview

OpenH323 and OPAL are protocol libraries that are intended to scale from one to as many connections as can be supported by the host hardware. The architecture of these libraries was established in 1999 when commodity hardware was capable of supporting a few hundred connections at most. In this environment, clustering of machines was the recommended approach for handling high call densities.

Since that time, multi-processor and multi-gigahertz CPU speeds have become commonplace. These machines should be able to easily handle many hundreds of simultaneous connections, but the current library architecture reaches resource limits long before CPU or memory saturation occurs. While this is not a problem for applications that handle one or two simultaneous calls, it prevents the use of OpenH323 and OPAL in environment where high call densities are required. The expectation is that a modern machine should be able to handle a call density in excess of 1000 calls.

A new architecture is needed to allow the libraries to scale well on current hardware platforms. This white paper describes the current architecture and its limitations and advantages, and offers proposals for several new architectures that address the issues that are raised. The intent is for this document to elicit comment from the OpenH323 and OPAL community about this issue, so that implementation of a solution can proceed.

For the purposes of this document, the name OpenH323 is used to describe both OPAL and OpenH323 as both libraries have the same call handling architecture. This document also concentrates on the call handling model for H.323 calls, as the SIP call model is similar but uses less resources due to its need to multiplex calls through a single UDP socket.

3. Current architecture

OpenH323 is based on the “one socket, one thread” model. This model gives every socket it’s own handling thread that spends most of the time blocked and waiting for data to arrive.

Every H.323 call handled by OpenH323 uses at least one socket for the H.225 channel. Every H.225 socket has an associated controlling thread. Another socket, with controlling thread, is required if H.245 tunneling is not used. Two more sockets are required if the connection uses the local RTP stack, with audio and video each needing a separate RTP stack with two sockets each. Two threads are also associated with each RTP session, but they are allocated to the incoming and outgoing sessions, rather than one per socket. This shows that each H.323 audio connection uses up to four sockets and four threads, with an audio/video call using six sockets and six threads.

The “one thread, one socket” model was chosen in 1999 on the basis of the following assumptions:

1. Threads are cheap, efficient and scale well
2. The most efficient form of blocking for a thread is I/O blocking
3. Sequential code is easier to write than state machines
4. Having many sockets in a single “select” system call is inefficient due to the need to maintain a linear list of socket IDs (the `fd_set` parameter)
5. The system will run out of resources for handling calls before it runs out of threads

We will examine each of these assumptions.

Threads are cheap and efficient

The two operating systems with the majority market share (Windows and Linux) have extensive support for lightweight threads. In particular, Linux has made enormous advances in the performance and usability of threads since the introduction of the NPTL (New Posix Thread Library) where threads were finally supported by the kernel.

While there is no intrinsic limit on the number of the threads that a Linux process can support, the PwLib support for Linux threads is limited to an absolute maximum of approximately 500 threads per process, which is imposed by the need for PwLib to allocate a Unix pipe (using two file handles) for each thread. This pipe is used to break out of socket read operations if the socket is closed pre-emptively, and also to restart paused threads. The limit arises due to the per-process file handle limit (which has a default limit of 1024) and the fact that PwLib uses a naïve allocation algorithm that statically allocates a pipe pair (consuming two file handles) to every thread at the time it constructed.

This requirement for pipe for every thread combines with the per-call socket and thread requirements to place an upper limit on the number of H.323 calls that a single Linux process can handle. This limit is determined as follows:

$$C_{Linux} = (H - H_{overhead}) / (T * 2 + S)$$

Where:

C_{Linux}	= total number of calls that can be created by a single Linux process
H	= per-process file handle limit, which is 1024 for a standard kernel
$H_{overhead}$	= number of handles used by OpenH323, which is 6
T	= number of threads per call ¹
S	= number of sockets per call

¹ The number of threads per call ignores the de-jitter thread that may be necessary if the audio is being transferred to a hardware device for human consumption in real-time. This can be ignored safely because these applications almost always have call limits imposed by the number of ports available on the hardware device that are reached well before any software limits.

The following table shows the values of C for all combinations of H.245 tunnelling and RTP usage on a Linux host (note that all of the calculations are based on the usage model for an audio call, although similar numbers can be run for audio/video calls):

H.245 tunnelling	RTP Stack	T	S	C_{Linux}
No	Yes	4	4	84
Yes	Yes	3	3	113
No	No	2	2	169
Yes	No	1	1	339

Figure 1 - Maximum H.323 audio calls per process on Linux with 1024 file handles

Note that the per-process file handle limit can be expanded on Linux to 16384 with only minor tweaking. This gives the following:

H.245 tunnelling	RTP Stack	T	S	C_{Linux}
No	Yes	4	4	1364
Yes	Yes	3	3	1819
No	No	2	2	2729
Yes	No	1	1	5459

Figure 2 - Maximum H.323 audio calls per process on Linux with 16384 file handles

While this appears give suitably large values for C_{Linux} , this calculation does not take into consideration the Linux per-process thread limit of 1024. In this regime, C_{Linux} is calculated as follows:

$$C_{Linux} = (L - L_{overhead}) / T$$

Where:

- C_{Linux} = total number of calls that can be created by a single Linux process
- L = per-process thread limit, which is normally 1024
- $L_{overhead}$ = number of threads used by OpenH323, which is 3
- T = number of threads per call

This gives the following:

H.245 tunnelling	RTP Stack	T	C_{Linux}	Total sockets used
No	Yes	4	255	1020
Yes	Yes	3	340	1020
No	No	2	510	1020
Yes	No	1	1022	1022

Figure 3 - Maximum H.323 audio calls per process on Linux with 1024 threads

In all cases, the total sockets used is below the per-process file handle limit, indicating that the maximum per-process call density on Linux is constrained by the per-process threads limit.

The Windows operating system has a per-process limit of around 2000 threads if the default 2Mbyte per-process stack size is used. The number of threads can be increased if the stack size is reduced, up to a practical maximum of about 10000-13000 threads but this depends on the memory consumption of each call thread. There is no documented per-process socket limit for Windows, but some informal tests by the author show that Windows 2000 allows the creation of over 20000 sockets by a single process.

Given these numbers, the maximum number of calls for a Windows process is related to the per-process thread limit (L) as follows:

$$C_{Windows} = (L - L_{overhead}) / T$$

Where:

$C_{Windows}$ = total number of calls that can be created by a single Windows process
 L = per-process thread limit, which is normally 2000
 $L_{overhead}$ = number of threads used by PWLib, which is 1
 T = number of threads per call

The following table shows the values of $C_{Windows}$ for all combinations of H.245 tunnelling and RTP usage on a Windows host:

H.245 tunnelling	RTP Stack	T	$C_{Windows}$	Total sockets used
No	Yes	4	499	1996
Yes	Yes	3	665	1995
No	No	2	998	1996
Yes	No	1	1996	1996

Figure 4 - Maximum H.323 audio calls per process on Windows

In all cases, the total sockets used is below the per-process file handle limit, indicating that the maximum per-process call density on Windows is constrained by the per-process threads limit.

From personal observation, the call densities described above are not sufficient to saturate modern host computers with a single OpenH323 process. Even multiple OpenH323 processes are usually insufficient, especially if multi-processor systems are used. This would tend to indicate that threads are an efficient way to utilise processor resources, even in numbers that approach the operating system limits, and that the fundamental assumption that threads are “cheap and efficient” still holds even if the total number of threads is capped by other resources.

A side-effect of the extensive use of threads within OpenH323 is the ability for the host operating system to automatically distribute load across multiple CPUs if they are present. Both Linux and Windows have the capability, and this provides exceptional load balancing capability at zero cost to the applications programmer.

Thread creation, deletion and rescheduling times do not appear to be significant problem for modern Linux kernels or Windows systems, although this was not always the case².

I/O blocking is efficient

The OpenH323 library relies on I/O blocking to drive the real-time aspects of the protocol stack. All of the threads involved in a call are composed of a control loop that blocks to read or write data from or to an I/O device, perform whatever processing is required, and then repeats. The loop is terminated by various error conditions or when an external thread closes the device. This formula is used very frequently in OpenH323 and is referred to as the “open/read/process/loop-until-closed” paradigm.

The H.225, H.245 and incoming RTP threads all block on reading data from a network socket. The outgoing RTP thread blocks on reading data from the audio capture device, which could be a sound card or another RTP connection. If a jitter buffer is being used to reassemble incoming audio data, then that thread blocks on writing data to the audio output device.

Personal experience indicates that modern operating systems such as Linux and Windows are very efficient at de-scheduling and rescheduling large numbers of threads that are blocked on I/O. This is confirmed by observing that a host computer running multiple OpenH323 processes at maximum call densities consume no noticeable CPU when there is no network traffic.

Rescheduling latency has not been seen as a problem, although there are often problems caused by interaction between threads including deadlock, priority inversion, and thread starvation.

² Windows 98 could be easily saturated with even a moderate number of threads, which could produce thread creation times on the order of 5-10 seconds. This does not appear to be an issue with Windows 2000 or Windows XP. Linux kernels prior to the introduction of NPTEL experienced similar problems, but there are now well-documented benchmarks showing that recent kernels are able to create and destroy 100000 threads in 2 seconds.

Sequential code is easier to write than state machines

The naïve implementation of a network protocol stack consists of sequential code containing network reads and writes interspersed with switching logic that implements the protocol. This approach lends itself to rapid prototyping and debugging as the code tends to follow the same control flow as the original network protocol specification

The sequential coding approach does not lend itself to handling multiple simultaneous connections as each read handles only a single connection. This leads to the paradigm encouraged by the Unix “select” function, where the code consists of one “select” function call that handles dozens or even hundreds of network connections, surrounded by logic that implements the network protocol as a state machine.

The personal experience of the OpenH323 authors is that state machines written using the “select” model tends to be harder to write and maintain than sequential code, which needs to be balanced against the fact that “select” driven code is usually more efficient in terms of memory and thread resources, especially if multiple worker threads are used to distribute the load and reduce latency.

At the time that the OpenH323 project was started, the authors were concerned about the complexity of the H.323 protocol and the maintenance of OpenH323 in the long term. This was seen as a higher priority than efficiency in high-density environments where memory or CPU resources might be constrained. As result, the decision was made to write the H.225 and H.245 handlers as sequential code so that they were as simple as possible. This was intended to reduce the likelihood of errors caused by hidden state transitions.

It is easy to look at the current code and observe that the handlers for H.225 and H.245 can be easily rewritten to use the “select” model. This is due to each PDU being contained within a TKT wrapper that allows it to be received in a single read³ and then processed. In fact, the GnuGK project has used this approach using it’s own internal socket handling routines.

However, this statement is 20-20 hindsight at it’s worst – this outcome was not obvious in 1999 and by the time it became obvious, there were other development pressures that prevented this architectural change from being implemented. Regardless, it is true that the current code-base lends itself to using the “select” model if required with very few changes.

Efficiency of “select”

The “select” function uses a bitmap to indicate which network connections should be checked for availability. The numeric value of the file handle is used as the bitmap index, which means that the bitmap may be quite large even if it contains only a few files handles. This can be compared to the “poll” system function that uses a list of file handles with an event mask.

The trade-off between “select” and “poll” is a subject of wide and varied debate, but the general consensus is that “poll” is better for file handle sets that are sparse and contain few members, while “select” is better for file handle sets that are dense and contain many members.

In either case, the handling of the file handle set is a crucial factor in determining the efficiency of “select” or “poll” particularly when it contains many members. The current OpenH323 design avoids this issue by always using “select” with (at most) one or two members.

³ Two reads, really. The bytes containing the length are read first, and then this is used to read remainder of the TPKT. But this does not change the argument

System resources vs thread count

The time since the creation of OpenH323 has seen Moore's Law reverse the pattern of resource usage for H.323 calls. A H.323 call requires the same amount of computing power as it did six years ago, but Moore's Law doubles the available power every 18 months. This means that we can assume an OpenH323 process has 16 times the available computing power than it did in 1999.

It follows that the original assumption that every socket needs it's own thread to provide sufficient performance is no longer valid. Increases in compiler efficiency, and improvements to the OpenH323 code base will tip the balance even further.

4. Analysis

Thread to call ratio

The previous section of this document reveals that the per-process limit on calls is related to the per-process limit on threads, provided that the per-process limit on sockets is suitably large. The relationship between the thread limit and the call limit can be approximated as follows:

$$C = L / T$$

Where:

C	= total number of calls that can be created by a single process
L	= per- process thread limit (for Windows this has the value 2000. For Linux this has the value 1024)
T	= threads per call

The current architecture implements values of T between 1 and 4. The table below shows possible values of C for various values of T , along with the maximum and minimum socket usage.

C_{Linux}	T	Sockets used	
		4 socket / call	1 socket / call
250	4	1000	250
333	3	1333	333
512	2	2048	512
1024	1	4096	1024
2048	0.5	8192	2048
4096	0.25	16384	4096
10240	0.1	40960	10240
20480	0.05	81920	20480

Figure 5 – Per-process call capacity for L = 1024 on Linux

This indicates that a T value of 1 provides a worst-case per-process call capacity 1024 calls with maximum socket usage of 4096 sockets. The Linux per-process socket limit of 16384 occurs with a T value of 0.25 (for 4 sockets calls) providing 4096 simultaneous calls. Single socket calls reach saturation at a T value of approx 0.08.

$C_{Windows}$	T	Sockets used	
		4 sockets / call	1 socket / call
500	4	2000	500
666	3	2666	666
1000	2	4000	1000
2000	1	8000	2000
4000	0.5	16000	4000
8000	0.25	32000	8000
20000	0.1	80000	20000
40000	0.05	160000	40000

Figure 6 - Per-process call capacity for L = 2000 on Windows

This indicates that a T value of 1 provides a worst-case per-process call capacity 2000 calls on Windows with maximum socket usage of 8000 sockets. The Windows per-process socket limit of 16384 occurs with a T value of approx 0.5 (for 4 sockets calls) providing 4000 simultaneous calls. Single socket calls reach saturation at a T value of approx 0.15.

The above analysis shows that there are two crucial value of T for each platform, as follows:

$1 < T$	C limited by per-process thread limit
$0.25 < T < 1$	C for 4 socket calls limited by per-process socket limit C for 1 socket calls limited per-process thread limit
$T < 0.25$	C limited by per-process socket limit

Figure 7 – Relationship between C and L for Linux

$1 < T$	C limited by per-process thread limit
$0.5 < T < 1$	C for 4 socket calls limited by per-process socket limit C for 1 socket calls limited per-process thread limit
$T < 0.5$	C limited by per-process socket limit

Figure 8 - Relationship between C and L for Windows

Call thread behaviour

The previous section has demonstrated that the OpenH323 per-process call limit could be improved by allowing a thread to handle more than one call. An obvious solution is to modify the call threads to use a “select” based model, this allowing a single thread to handle a large number of calls. The H.225 or H.245 threads appear to be particularly good candidates for this treatment.

However, there are other aspects of the call threads that make this naïve approach less attractive than might first appear.

H.225 and H.245 threads

The threads processing the messages on the H.225 and H.245 sockets (if tunnelling is not used) are very similar. The top level implementation of these threads are examples of the “open/read/process/loop-until-closed” paradigm⁴.

The messages handled by these threads are few in number (around 5 messages received per call), and are mostly sent used during call setup. Once the call is connected, the H.225 and H.245 channels carry very little traffic, being used primarily for sending out of band DTMF (user indication messages).

These two threads handle most of the protocol-related functionality for a call during the time between call start, and call answer. The vast majority of the virtual functions available on the H323Connction and H323EndPoint classes⁵ are called in the context of these threads. The current architecture of “one socket, one thread” allows the developer to effectively ignore the time taken to execute virtual functions related to the protocol because each call has it’s own thread separate from every other call.

This robustness is lost if multiple H.225 or H.245 calls are aggregated into a single thread, because any delay in a virtual function for one call will delay the handling of every other call that uses the same aggregator thread. Even worse, any application that implements synchronous communications between the callbacks of two different calls (this is a common architecture used for proxies) will deadlock if the same thread handles these two calls.

The vast majority of the opportunities for calls to deadlock or delay each other occurs during the exchange of protocol messages prior to a call being answered, and during the capability exchange just after call answer and while opening the media channels. Once this period has been completed, the call threads handle very little traffic and are excellent candidates for aggregation.

When a call shutdown is triggered, OpenH323 closes the network connections and aggregates the calls into a single cleanup thread.

⁴ See the `H323Connection::HandleSignallingChannel()` and `H323Connection::HandleControlChannel` in `h323.cxx`

⁵ Or the OPAL equivalents of `OpalConnection` and `OpalManager`

RTP read thread

The RTP read thread is another example of the “open/read/process/loop-until-closed” paradigm ⁶

This thread handles a continuous stream of incoming media for the duration of the call. The media may not be very large, but it is essential that the thread maintain sufficient data throughput to avoid adding jitter into the media stream. The audio quality of the call is severely compromised if the read thread is unable to maintain adequate data throughput.

The incoming data from the RTP ports is handled in one of two ways depending on whether the media stream needs to be de-jittered or not. Two threads are used to handle media that needs de-jittering, such as audio being written to a local hardware device. One thread blocks on reading from the network sockets and writes media into a jitter buffer, while the second thread reads from the jitter buffer and blocks on writing data to the audio hardware device. Both of these threads derive their timing from I/O blocking (one read and one write) but only the first thread is based on socket I/O that makes it suitable for aggregation with other similar threads via a “select” function.

Media that does not need de-jittering - such as video or audio being proxied to another network connection – is handled by one thread that reads from the RTP ports and writes to the output hardware device or network connection. The thread is required to block on the socket read in order to ensure data is transferred at the correct rate. The write device is not required to block, but may do so if desired. If it does block, then the thread becomes “lock-stepped” as it reads and writes between the two devices that are creating and consuming data at the same rate.

This aspect of the single thread version of the RTP read thread means that it is not suitable for aggregation. To understand this, consider the case where multiple RTP read threads are aggregated and where one of the threads uses a write device that blocks. This device will drive the timing for all RTP sessions that are aggregated into the thread, ensuring that data is not consumed any faster than that one device can accept it.

RTP write thread

The RTP write thread reads from the media source device and writes to the RTP data sockets. These threads are not suitable for aggregation because the timing is driven by the media source device that are not guaranteed to be compatible with the “select” function call.

Summary

An above examination of the threads involved in a call reveals the following:

- The H.225 and H.245 call threads are not suitable for aggregation until terminal capability sets have been acknowledged, the media channels have been opened, and the call has connected. The network connections can then be aggregated as they handle very little traffic.
- When de-jittered media is being used, the RTP read thread that reads from the RTP network sockets is suitable for aggregation provided that the real-time constraints for RTP can be maintained. The jitter buffer read thread, and the RTP read thread used for non-de-jittered media are not suitable for aggregation.
- The RTP write thread is not suitable for aggregation as it derives its timing from a device.

⁶ See `RTP_UDP::ReadData` in `rtp.cxx`

5. Requirements specification

Two modifications are proposed that will decrease the number of threads used for each process:

1. The H.225 and H.245 call threads shall be modified to use thread aggregation after a call has completed the terminal capability set exchange, or some time after the call has connected, whichever happens last. Each call thread aggregator will handle a maximum of 25 calls.
2. The RTP read threads shall be modified to automatically aggregate if the device requires jittering. Each RTP thread aggregator will handle a maximum of ten calls.

These changes will implement an average T value of around 0.04 if calls are signalling only. This will give a maximum value of 25000 for C_{Linux} and 50000 for $C_{Windows}$.

Calls using RTP will still require an RTP write thread for every call, but these modifications will allow a T to approach a value of 1. This will give a maximum value of 1000 for C_{Linux} and 2000 for $C_{Windows}$.

This section below provides a functional specification of the code changes that are proposed to implement these modifications.

Call thread

- 5.1.1 The existing architecture whereby each H.323 call uses separate dedicated threads to handle H.225 and H.245 messages for each call shall be retained.
 - 5.1.1.1 The separate call threads shall exist from the time of call creation until a point later in the call lifetime where certain specific conditions are met, at which time the network sockets shall be “aggregated” to reduce the overall thread count.
- 5.1.2 A member variable of type BOOL on the H323Connection object shall be added to control whether a specific call is permitted to participate in H.225 and H.245 aggregation
 - 5.1.2.1 The member variable of H323Connection described in 5.1.2 shall be initialized during the class constructor to the value of a member variable of the H323Endpoint class to allow the default value to be changed easily
 - 5.1.2.2 The default value for the member variable of H323Endpoint described in 5.1.2.1 shall be set to “TRUE”, which will allow aggregation to take place
- 5.1.3 A H.225 or H.245 call thread that is unique to a call shall automatically transfer the H.225 and H.245 sockets to an “aggregation thread” for handling provided all of the conditions described below have been met:
 - 5.1.3.1 The call is permitted to participate in call aggregation (see 5.1.2)
 - 5.1.3.2 The call has received and sent terminal capability set acknowledgments
 - 5.1.3.3 A CONNECT PDU has been send or received
 - 5.1.3.4 The media channels have been opened
- 5.1.4 The maximum number of H.225 or H.245 sockets that can be aggregated into a single thread shall be set by a member variable of the H323Endpoint class of type PINDEX
 - 5.1.4.1 The default value for the number of H.225 or H.245 sockets that can be aggregated into a single thread shall be set to “25”
- 5.1.5 The following shall apply to the transfer of H.225 and H.245 sockets to an aggregation thread:

- 5.1.5.1 A new aggregation thread shall be created if no thread is available when a call meets the conditions specified in 5.1.3.
- 5.1.5.2 A call shall always transfer sockets to the aggregation thread that has the minimum number of active sockets.
- 5.1.5.3 A new aggregation thread shall be created if all existing aggregation threads contain the maximum number of sockets (see 5.1.4.1)

RTP read thread

- 5.2.1 The existing architecture whereby each H.323 call uses separate dedicated threads to handle RTP media for each call shall be retained except for calls that meet specific criteria, whereupon the RTP networks connections shall be “aggregated” to reduce the overall thread count.
- 5.2.2 A member variable of type BOOL on the H323Connection object shall be added to control whether a specific call is permitted to participate in RTP read thread aggregation
 - 5.2.2.1 The member variable of H323Connection described in 5.2.2 shall be initialized during the class constructor to the value of a member variable of the H323Endpoint class to allow the default value to be changed easily
 - 5.2.2.2 The default value for the member variable of H323Endpoint described in 5.2.2.1 shall be set to “TRUE”, which will allow aggregation to take place
- 5.2.3 The stack shall be modified to allow the aggregation of RTP read threads under the following conditions:
 - 5.2.3.1 The call is permitted to participate in RTP read thread aggregation (see 5.2.2)
 - 5.2.3.2 The RTP thread is reading media that requires de-jittering
- 5.2.4 The maximum number of RTP sockets that can be aggregated into a single thread shall be set by a member variable of the H323Endpoint class of type PINDEX
 - 5.2.4.1 The default value for the number of H.225 or H.245 sockets that can be aggregated into a single thread shall be set to “10”
- 5.2.5 The following shall apply to the transfer of RTP sockets to an aggregation thread:
 - 5.2.5.1 A new aggregation thread shall be created if no thread is available when a call meets the conditions specified in 5.2.3.
 - 5.2.5.2 A call shall always transfer sockets to the aggregation thread that has the minimum number of active sockets.
 - 5.2.5.3 A new aggregation thread shall be created if all existing aggregation threads contain the maximum number of sockets (see 5.2.4.1)

6. Conclusion

The existing OpenH323 architecture does not achieve the desired call densities on modern hardware platforms, because the “one socket, one thread” model reaches the per-process thread limit before CPU saturation or memory size constraints is achieved.

The modifications proposed will allow the call density to approach 1000 for calls including RTP, and 25000 for calls that are signaling only.

Thanks to the following people for their assistance in the preparation of this document:

Derek Smithies
Louis R. Marascio
Peter Nixon
