



THE UNIVERSITY OF QUEENSLAND
A U S T R A L I A

Title here:

Name

Candidate's academic degrees



Candidate's ORCID

*A thesis submitted for the degree of Doctor {Master} of Philosophy at
The University of Queensland in {year}*

Name of the Enrolling Unit

Abstract

Start this section on a new page [this template will automatically handle this].

The abstract should outline the main approach and findings of the thesis and normally must be between 300 and 800 words.

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, financial support and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my higher degree by research candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis and have sought permission from co-authors for any jointly authored works included in the thesis.

Acknowledgments

Contents

Abstract	ii
Contents	iv
List of Figures	vi
List of Tables	vii
List of Abbreviations and Symbols	ix
1 Introduction	3
1.1 Motivation	3
1.2 Aims and Scope	3
2 Lit. Review	5
2.1 Computer Networks, TCP/IP	5
2.1.1 Link Layer and Ethernet PHY	5
2.2 Network Security	5
2.2.1 Encryption	5
2.2.2 Performing an AES encryption via OpenSSL	7
2.3 RISC-V Instruction Set Architecture	7
2.4 FPGAs	8
2.4.1 Softcore CPUs	8
2.4.2 Chosen Softcore CPU: VexRiscv	8
2.4.3 SpinalHDL	8
2.4.4 CPU Buses, Wishbone	8
2.4.5 VexRiscvSMP	8
2.5 Litex	8
2.5.1 Cores	8
2.5.2 Litex BIOS	8
2.5.3 Buildroot Linux	8

3 Evaluation	9
3.1 Raw Performance Benchmarks	9
3.1.1 Iperf3	9
3.1.2 Stress NG	10
3.2 Results & Analysis	12
3.2.1 Test Suite Full Outline	12
3.2.2 Single Core Results	13
3.2.3 Dual Core Results	14
3.2.4 Quad Core Results	15
3.2.5 Raspberry Pi Results	16
3.3 Improved Dual Core Design, Results and Analysis	16
3.4 Utilisation	16
3.5 Timing	16
3.6 Power Usage	16
Bibliography	17
A Appendix	19
A.1 Full Single Core Results	20
A.2 Full Dual Core Results	23
A.3 Full Quad Core Results	26
A.4 Full Raspberry Pi Results	29
A.5 Full Improved Dual Core Results	32
A.6 Name of Appendix-2	35

List of Figures

2.1	AES Round Function [1]	6
3.1	Stress-ng Performance Comparison Across Configurations	11
A.1	Single Core, Basic Test 1 Server Metrics	20
A.2	Single Core, Basic Test 2 Server and Client Metrics	20
A.3	Single Core, Basic Test 3 Server and Client Metrics	21
A.4	Single Core, Large Scale, 100 clients, Server and Client Metrics	22
A.5	Dual Core, Basic Test 1 Server Metrics	23
A.6	Dual Core, Basic Test 2 Server and Client Metrics	23
A.7	Dual Core, Basic Test 3 Server and Client Metrics	24
A.8	Dual Core, Large Scale, 100 clients, Server and Client Metrics	25
A.9	Quad Core, Basic Test 1 Server Metrics	26
A.10	Quad Core, Basic Test 2 Server and Client Metrics	26
A.11	Quad Core, Basic Test 3 Server and Client Metrics	27
A.12	Quad Core, Large Scale, 100 clients, Server and Client Metrics	28
A.13	Raspberry Pi, Basic Test 1 Server Metrics	29
A.14	Raspberry Pi, Basic Test 2 Server and Client Metrics	29
A.15	Raspberry Pi, Basic Test 3 Server and Client Metrics	30
A.16	Raspberry Pi, Large Scale, 100 clients, Server and Client Metrics	31
A.17	Improved Dual Core, Basic Test 1 Server Metrics	32
A.18	Improved Dual Core, Basic Test 2 Server and Client Metrics	32
A.19	Improved Dual Core, Basic Test 3 Server and Client Metrics	33
A.20	Improved Dual Core, Large Scale, 100 clients, Server and Client Metrics	34

List of Tables

3.1	Ethernet Throughput Comparison of Configurations	9
3.2	Stress-ng Comparison of Configurations	10

List of Abbreviations and Symbols

Abbreviations

AC	Alternating Current
AFM	Atomic Force Microscopy/Microscope
<i>etc.</i>	<i>etc.</i>

Symbols

$\hat{\rho}$	Density operator
<i>etc.</i>	<i>etc.</i>

List of todos

Add reference to software/hardware overview	9
make reference to hardware overview	11
Make reference to software setup	12
add reference to proof of encryption	12
Make reference to database	12
Make reference to schema	12

Chapter 1

Introduction

1.1 Motivation

1.2 Aims and Scope

Chapter 2

Literature Review

2.1 Computer Networks, TCP/IP

2.1.1 Link Layer and Ethernet PHY

2.2 Network Security

As more applications turn to IoT edge-computing, it is critical that engineers ensure that computer networks still remain secure despite resource limitations. Key considerations for network transfers include implementing cryptography and standard security protocols. Unfortunately, the more we implement, the more complicated our applications become, straining edge computers which are most efficient when they are only single-purpose.

Additionally, it's not only network transfers where vulnerabilities can exist. Physical security measures must also be considered, as many embedded systems are deployed in accessible locations. Although unlikely, it is possible for dedicated hackers to create intrusions and RCEs based on memory-tampering. Such attacks have been around since the conception of the C programming language, Shao *et al.* shows how these attacks can be mitigated at a higher-level [2], but ultimately, the onus is on developers to ensure that all low-level accesses to memory are consistently safe; *i.e.*, there are no possible edge cases that can trigger buffer overflows.

We can address these issues, and still have robust security posture for networked embedded systems, however, to be implemented properly, regular reviews and updates are also required as the field of cybersecurity is constantly adapting to new vulnerabilities.

2.2.1 Encryption

This thesis will focus on efficiently implementing one such consideration, albeit a critical one, which is encryption. Specifically, we will be implementing the Advanced Encryption Standard (AES), which has become the de-facto standard for symmetric key encryption in computer security.

At its core, AES is a block cipher that operates on fixed-size blocks of data (128 bits) using cipher keys of 128, 192, or 256 bits. The encryption process consists of multiple rounds of several processing steps, all of which aim to obfuscate the input data, yet also maintain reversibility for decryption. Here is a high-level overview of the specific steps [1]:

High-Level Overview of the AES Algorithm

1. *KeyExpansion* – round keys are derived from the cipher key. AES uses a separate 128-bit round key block for each round plus one more.
2. *AddRoundKey* - each byte of the state is combined with a byte of the round key using bitwise XOR. This is how the key gets incorporated into the encryption process.

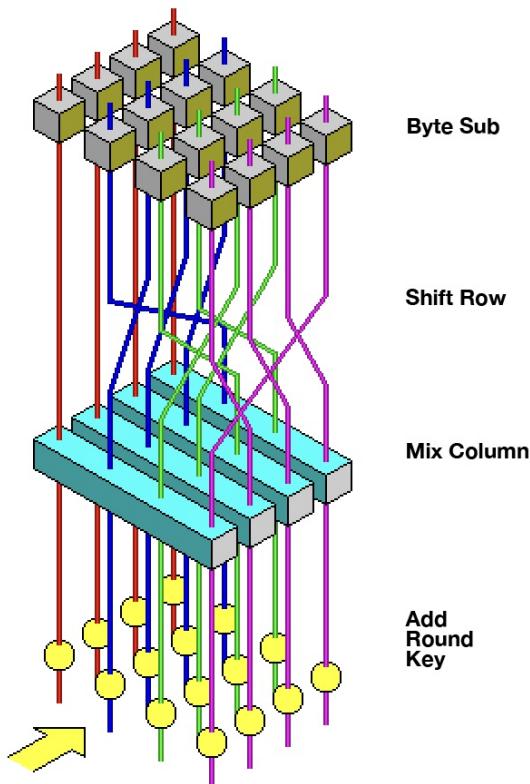


Figure 2.1: AES Round Function [1]

3. Middle rounds:

1. *SubBytes*: each byte is substituted with another according to a lookup table (the SBox).
2. *ShiftRows*: then, the last three rows of the substituted bytes are shifted cyclically.
3. *MixColumns*: The four bytes in each column are then combined.
4. *AddRoundKey*: finally, the round key is XOR-ed with each byte of the state (same as step 2).

These middle rounds are executed n times, depending on the key size:

- $n = 10$ rounds for 128-bit keys
- $n = 12$ rounds for 192-bit keys
- $n = 14$ rounds for 256-bit keys

The final output is our encrypted input text. AES has been heavily and publicly scrutinised for decades since its adoption by the U.S. government [3] (aside from side-channel attacks, not technically a fault of the algorithm). It's also performant, as it can be implemented in-place with constant-time operations. And it's flexible, as multiple key-sizes are available and can be chosen based on an application's memory/time complexity constraints.

It's worth mentioning too that the decryption process essentially reverses the encryption process; but it uses inverse versions of the encryption steps, applied in a reverse order.

2.2.2 Performing an AES encryption via OpenSSL

An AES 256-bit CBC encryption of a text file can be performed via the open-source, OpenSSL library which implements SSL and TLS protocols for linux systems:

```
openssl enc -aes-256-cbc -salt -in plaintext.txt -out encrypted.bin -k
MySecretKey -pbkdf2
```

Where:

- **-aes-256-cbc:** Specifies the encryption algorithm (AES-256 in CBC mode).
- **--salt:** Adds a salt to the password for added security.
- **--in plaintext.txt:** The input file to be encrypted.
- **--out encrypted.bin:** The output file where the encrypted data will be stored.
- **--k MySecretKey:** The key used for encryption.
- **--pbkdf2:** Uses PBKDF2 (Password-Based Key Derivation Function 2) for key generation, which is more secure than the default (A warning for deprecated key generation usage will appear if this is omitted (OpenSSL 3.3.1)).

Of course, the "plaintext.txt" file is only securely encrypted if the key is not made public.

2.3 RISC-V Instruction Set Architecture

RISC-V is an open-source Instruction Set Architecture (ISA) developed at the University of California, Berkeley in 2010 [1]. It aims to provide a modern, modular, and extensible ISA for a wide range of computing devices. The RISC-V ecosystem has grown rapidly, with major tech companies and startups adopting the architecture. As of 2024, RISC-V processors have made substantial progress in performance, though they still lag behind the most advanced x86 and ARM designs in high-performance applications. For instance, the SiFive Performance P670 core, announced in late 2022, claimed performance comparable to Arm Cortex-A77 cores from 2019 [2].

RISC-V's main advantage, is in its open-sourceness and potential for customization. Many companies are leveraging RISC-V for specialized applications, such as AI accelerators, where the ability to add custom instructions can provide significant advantages [3]. While RISC-V is still maturing in the high-performance computing space, it has already found significant success in microcontrollers and embedded systems. As of 2023, RISC-V International reported that over 10 billion RISC-V cores have been shipped, primarily in microcontrollers and embedded devices [4]. As the ecosystem continues to evolve, with more software support and advanced implementations, RISC-V is poised to become an increasingly important player in the processor market across various computing domains.

2.4 FPGAs

2.4.1 Softcore CPUs

2.4.2 Chosen Softcore CPU: VexRiscv

2.4.3 SpinalHDL

2.4.4 CPU Buses, Wishbone

2.4.5 VexRiscvSMP

2.5 Litex

2.5.1 Cores

2.5.2 Litex BIOS

2.5.3 Buildroot Linux

Chapter 3

Evaluation

3.1 Raw Performance Benchmarks

Here, the different configurations of VexRiscv: single-core, dual-core and quad core; will be compared in terms of performance alongside the Raspberry Pi Model 4B 1GB. For each one, we will run the performance benchmark, *stress-ng*, which profiles the IO overhead and memory usage of multiple cores, along with *Iperf3*, to gauge ethernet throughput capabilities.

3.1.1 Iperf3

As ethernet is a vital component of the application, it makes sense to evaluate the capabilities of the link, especially the average bitrate we can expect. After running *iperf3 -s*, which sets the device as a server and listens, another Raspberry Pi was chosen from the cluster to act as a client via running:

```
iperf3 -c 192.168.1.50 -t 30 -i 1 -w 8K -P 1 -R
```

This begins a single-threaded (-P 1), client that sends and receives TCP transmissions to *192.168.1.50*, for 30 seconds, sampling the bitrate every second (-i 1). Most notably, it constrains the TCP window size (-w 8K), to 8Kb, which matches the current size of the board's TX or RX ethernet buffers, more closely resembling the stop-start transfers in our software setup. Here are the results:

Table 3.1: Ethernet Throughput Comparison of Configurations

	VexRiscvSMP, 100MHz			RPi
	Single	Dual	Quad	4B 1GB
Amount Transferred (MB)	31.5	35.5	42.0	1.13k
Amount Received (MB)	31.4	35.4	41.9	1.13k
Sending Bitrate (MBits/s)	8.78	9.90	11.7	323
Receiving Bitrate (MBits/s)	8.77	9.89	11.7	323
TCP Retransmissions	0	0	1	0

It is clear that the gigabit ethernet capabilities of the Raspberry Pi far outweigh the ethernet capabilities of the board, achieving 26x more throughput than that of the quad-core VexRiscvSMP.

Add reference to software/hard overview

Keep in mind as well, the Raspberry Pi is throttled because of the 8Kb window size we set. Additionally, the core amount does have an effect on the bitrate as the quad core is 1.8MBits/s faster than the dual core, which is 1.12MBits/s faster than the single core. However, an overall improvement of 32% across cores is basically negligible since the absolute performance is still far below what is required for what is essentially a high-speed ethernet application. It is clear from this benchmark alone that the bitrate will be a significant bottleneck for the design.

3.1.2 Stress NG

Stress-ng is a versatile benchmarking tool designed to stress test various components of a CPU. The command:

```
stress-ng --cpu $CORE_COUNT --io 2 --vm 1 --vm-bytes 128M --timeout 60s
--metrics-brief
```

Runs a set of tests in parallel. The rest of the arguments determine the amount of tests and the type: --cpu, creates CPU-intensive tasks equal to the core count; --io, creates two I/O-intensive tasks; and --vm, allocates and uses 128MB of virtual memory. This will evaluate for us how the system performs under combined CPU, I/O, and memory pressure, as well as how these metrics vary with the amount of cores. Additionally, the test will timeout after 60 seconds or until $6 \cdot \$CORE_COUNT$ CPU bogo ops have been completed. Here are the results:

Table 3.2: Stress-ng Comparison of Configurations

	VexRiscvSMP, 100MHz	RPi		
	Single	Dual	Quad	4B 1GB
CPU bogo ops	6	12	24	12,483
CPU real time (s)	125.34	119.09	121.30	60.03
CPU usr time (s)	78.32	164.00	378.16	146.85
CPU sys time (s)	0.01	0.12	0.09	0.03
CPU bogo ops/s (real time)	0.05	0.10	0.20	207.94
CPU bogo ops/s (usr+sys time)	0.08	0.07	0.06	84.99
IO bogo ops	21,794	31,190	27,819	440,066
IO real time (s)	60.00	60.01	60.00	60.00
IO usr time (s)	2.74	3.56	3.42	10.34
IO sys time (s)	25.71	44.44	66.44	48.56
IO bogo ops/s (real time)	363.22	519.76	463.65	7,334.31
IO bogo ops/s (usr+sys time)	766.05	649.79	398.21	7,472.11
VM bogo ops	2,280	3,053	2,464	617,668
VM real time (s)	61.92	62.54	61.54	60.16
VM usr time (s)	7.57	10.72	18.10	27.27
VM sys time (s)	7.60	14.93	18.32	6.78
VM bogo ops/s (real time)	36.82	48.82	40.04	10,266.29
VM bogo ops/s (usr+sys time)	150.30	119.03	67.66	18,143.58
Total time taken (s)	125.45	120.31	122.76	60.00

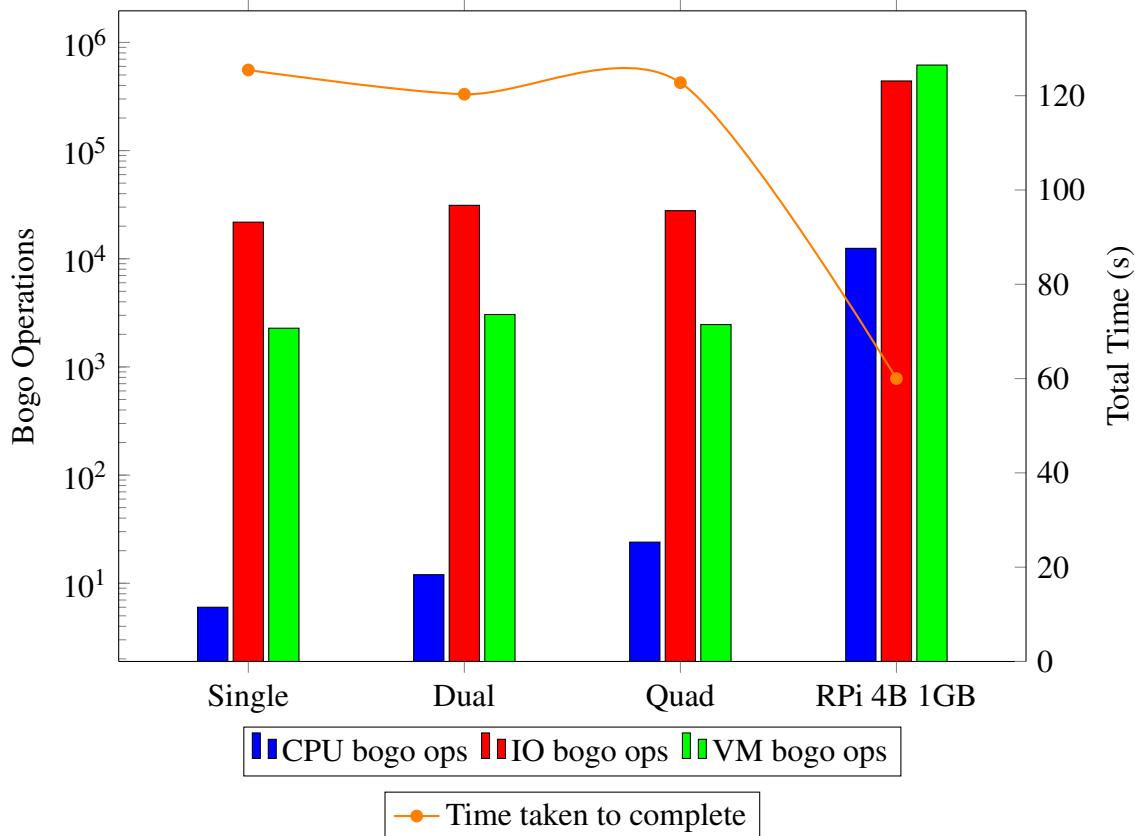


Figure 3.1: Stress-ng Performance Comparison Across Configurations

Immediately, we are presented with an extreme difference in performance scaling, which is to be expected when comparing an FPGA-based softcore processor to a Raspberry-Pi single-board computer. There is still nuance, however, between the different core configurations of the VexRiscvSMP. For instance, the amount of CPU bogo ops tend to scale linearly with the core count, maintaining a similar execution time (see "CPU real-time" in table: 3.2). Other than this, there are no clear trends for the IO or VM bogo operations. It seems that once we extend to four cores, there are resource contentions with the IO bus and memory, leading to the quad core performing slightly less favourably than the dual core in these operations. Therefore, IO and memory have the potential to be another two bottlenecks, in addition to the low ethernet throughput, (although ethernet is still the dominant bottleneck). This can give us an estimation of how the dual core will perform relative to the quad core, *i.e.*, it may end up performing better since the quad core will not only have to contend with IO/memory constraints from the shared DRAM, but also the limited ethernet buffers.

Overall, out of the VexRiscvSMPs, dual core performed the best, but only if we exclude raw CPU power. As our application depends more on resource read/writes and management, dual core may prove to be the best configuration; besides the Raspberry Pi of course.

make reference to hardware overview

3.2 Results & Analysis

In this section, we will now compare how each configuration performed running the exact same full suite of tests on our application. We will then take the best performing out of the VexRiscv configurations and see if we can improve the design to achieve even more throughput.

3.2.1 Test Suite Full Outline

The tests will focus only on encryption, since decryption and encryption times are the same, and we have proven that the device is encrypting properly. We are aiming to deduce how each configuration performs with single or multiple clients. Therefore, the test outline is as such:

1. Basic test 1:
1 client with 1 100mb file.
2. Basic test 2:
10 clients with varying file sizes: 2mb, 5mb, 10mb, 20mb.
3. Basic test 3:
20 clients with varying file sizes, in 10 client chunks.
4. Large scale test:
100 clients with varying files sizes, in 10 client chunks.

While these tests are running, we are collecting a multitude of metrics regarding overall performance, but the primary result we are concerned with is the total bytes processed (or bytes encrypted) per second. This single metric is all we need to determine which amount of cores is superior for the application. It is a standard average, calculated from:

$$\text{Average Bytes Processed}(\text{Bytes}/\text{s}) = \frac{\text{Total Bytes Processed}}{\text{Test Duration}}$$

Where '*Total Bytes Processed*' is the total amount of bytes processed during the test, and '*Test Duration*', is how long the test took to execute. This is calculated from the server-side, as all the information is present there.

On the client-side, we are also keeping track of the client's total time for the transaction, time spent in the queue, time spent on network transfers and the time spent processing on the server, i.e. encrypting. Total time, network time and queuing time are collected in real time on client-side, leaving the processing time as the remainder, calculated via:

$$\text{Processing Time}(s) = \text{Total Time} - \text{Network Time} - \text{Queue Time}$$

Lastly, all timings were collected using the '*Time*', Python standard library.

3.2.2 Single Core Results

Single Core Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	84.93	169.87	849.35
Avg Bytes per Second (Kb/s)	221.74	183.57	187.14	184.78
Test Duration (s)	472.89	462.69	907.70	4596.65
Total Network Time (s)	173.70	188.40	377.81	2100.95
Total Processing Time (s)	299.18	274.29	529.89	2495.71
Avg Queue Time (s)	0.15	8.28	7.66	8.05

3.2.3 Dual Core Results

Dual Core Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	84.93	169.87	849.35
Avg Bytes per Second (Kb/s)	272.97	265.26	272.79	264.22
Test Duration (s)	384.13	320.19	622.70	3214.54
Total Network Time (s)	152.47	110.49	256.35	1166.14
Total Processing Time (s)	231.66	209.70	366.36	2048.41
Avg Queue Time (s)	0.07	4.35	3.94	4.42

3.2.4 Quad Core Results

Quad Core Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	84.93	169.87	849.35
Avg Bytes per Second (Kb/s)	269.92	248.77	251.97	245.57
Test Duration (s)	388.48	341.42	674.16	3458.68
Total Network Time (s)	171.60	100.68	209.94	902.06
Total Processing Time (s)	216.88	240.74	464.23	2556.62
Avg Queue Time (s)	0.08	4.37	3.65	3.83

3.2.5 Raspberry Pi Results

Raspberry Pi Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	55.57	217.06	946.86
Avg Bytes per Second (Kb/s)	8042.64	13853.24	17895.55	16829.84
Test Duration (s)	13.04	4.01	12.13	56.26
Total Network Time (s)	8.06	2.05	8.13	37.50
Total Processing Time (s)	4.97	1.96	4.00	18.76
Avg Queue Time (s)	0.00	0.03	0.02	0.03

3.3 Improved Dual Core Design, Results and Analysis

Improved Dual Core Tests	Basic 1	Basic 2	Basic 3	Large Scale 100
Total Bytes Processed (Mb)	104.86	84.93	169.87	849.35
Avg Bytes per Second (Kb/s)	376.04	406.06	393.59	403.36
Test Duration (s)	278.85	209.17	431.59	2105.67
Total Network Time (s)	51.08	71.65	108.20	646.95
Total Processing Time (s)	227.77	137.52	323.39	1458.72
Avg Queue Time (s)	0.08	3.11	1.95	2.81

3.4 Utilisation

3.5 Timing

3.6 Power Usage

Bibliography

- [1] Wikipedia, Advanced encryption standard, https://en.wikipedia.org/wiki/Advanced_Encryption_Standard (Oct 2024).
- [2] Z. Shao, Q. Zhuge, Y. He, E.-M. Sha, Defending embedded systems against buffer overflow via hardware/software, in: 19th Annual Computer Security Applications Conference, 2003. Proceedings., 2003, pp. 352–361. doi:10.1109/CSAC.2003.1254340.
- [3] National Institute of Standards and Technology, Advanced encryption standard (aes), Federal Information Processing Standards Publication FIPS 197, National Institute of Standards and Technology, Gaithersburg, MD, updated May 9, 2023 (May 2023).
URL <https://doi.org/10.6028/NIST.FIPS.197-upd1>

Appendix A

Appendix

A.1 Full Single Core Results

Figure A.1: Single Core, Basic Test 1 Server Metrics

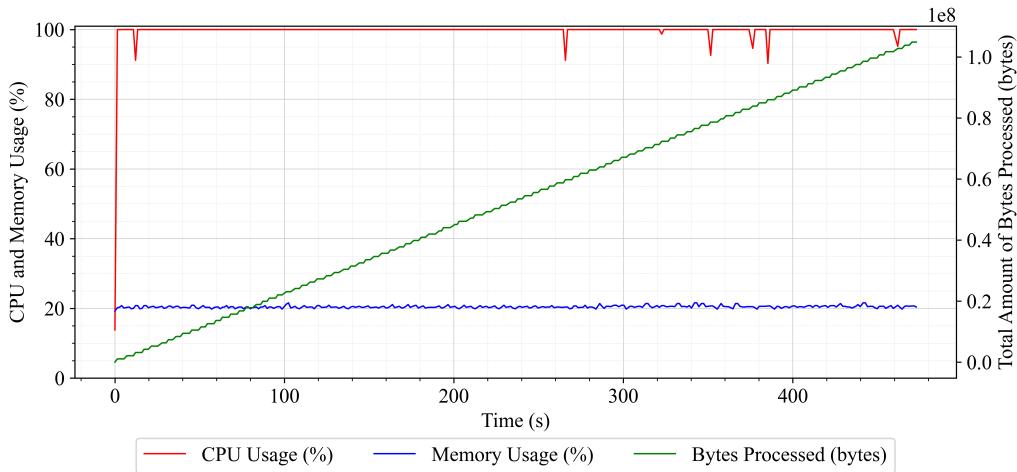


Figure A.2: Single Core, Basic Test 2 Server and Client Metrics

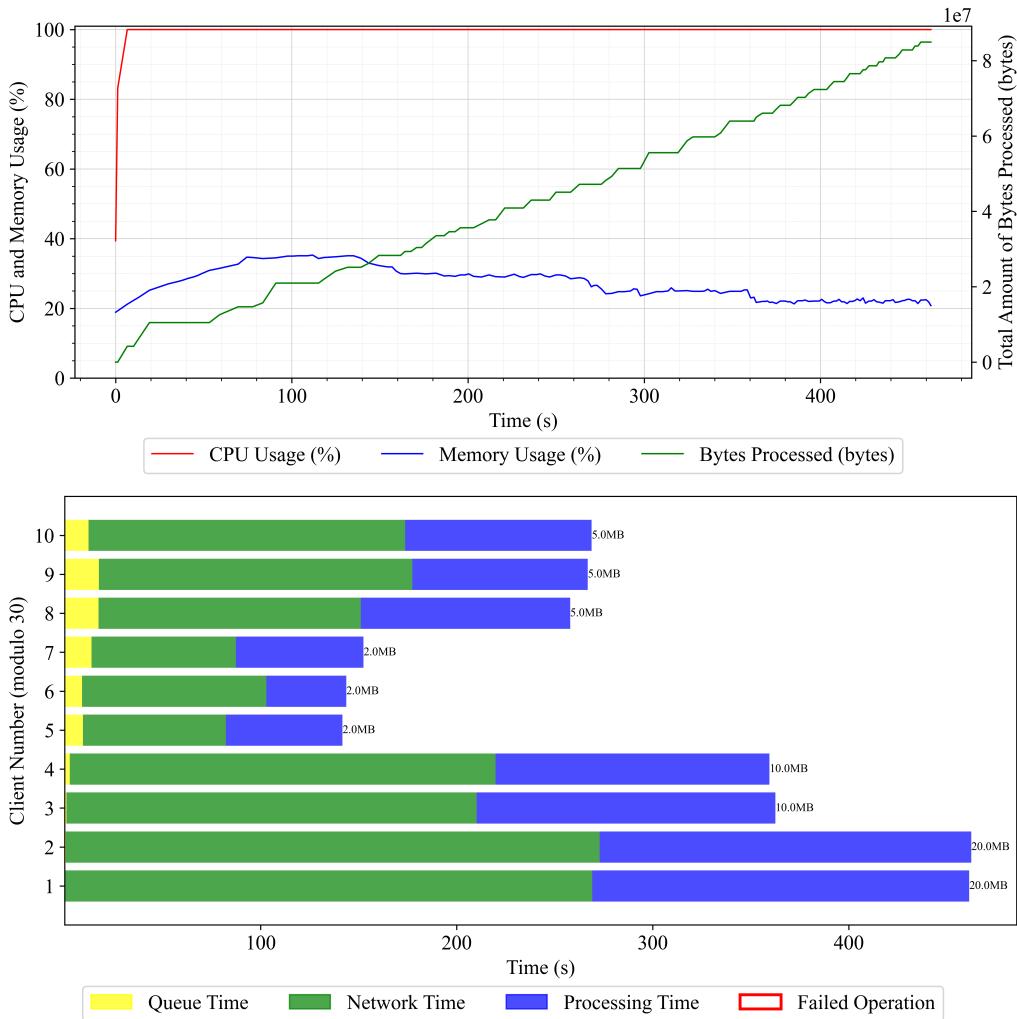


Figure A.3: Single Core, Basic Test 3 Server and Client Metrics

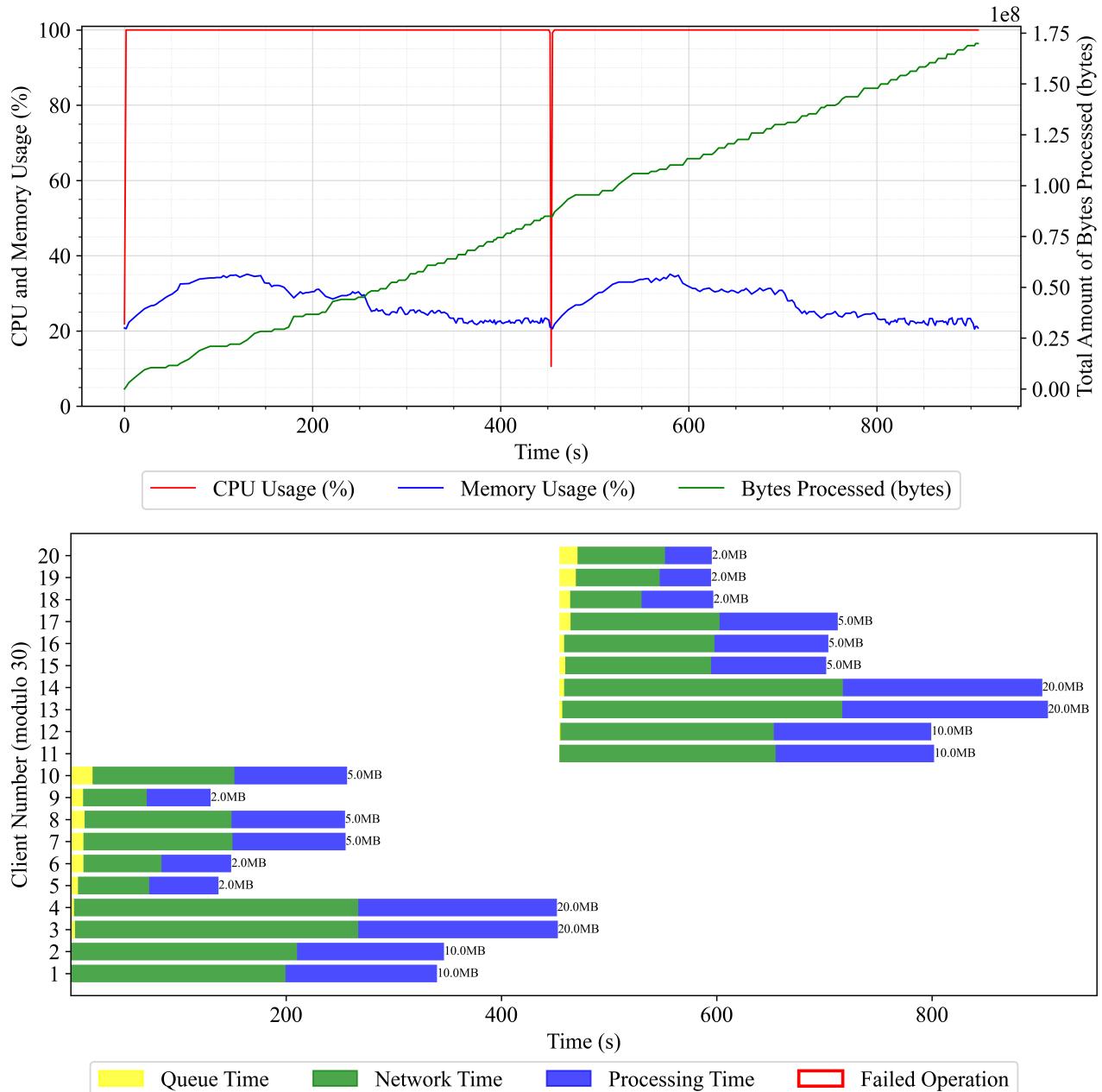
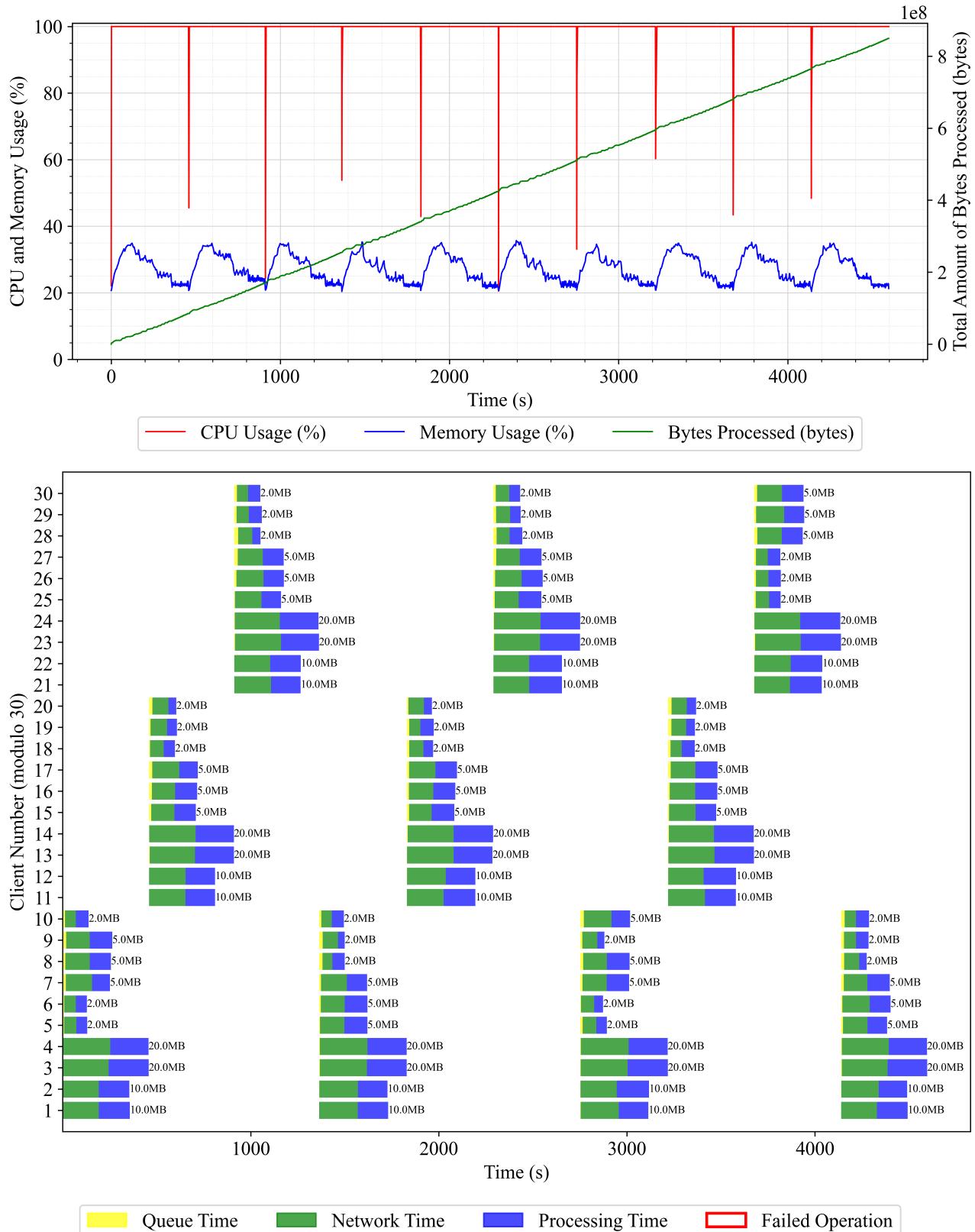


Figure A.4: Single Core, Large Scale, 100 clients, Server and Client Metrics



A.2 Full Dual Core Results

Figure A.5: Dual Core, Basic Test 1 Server Metrics

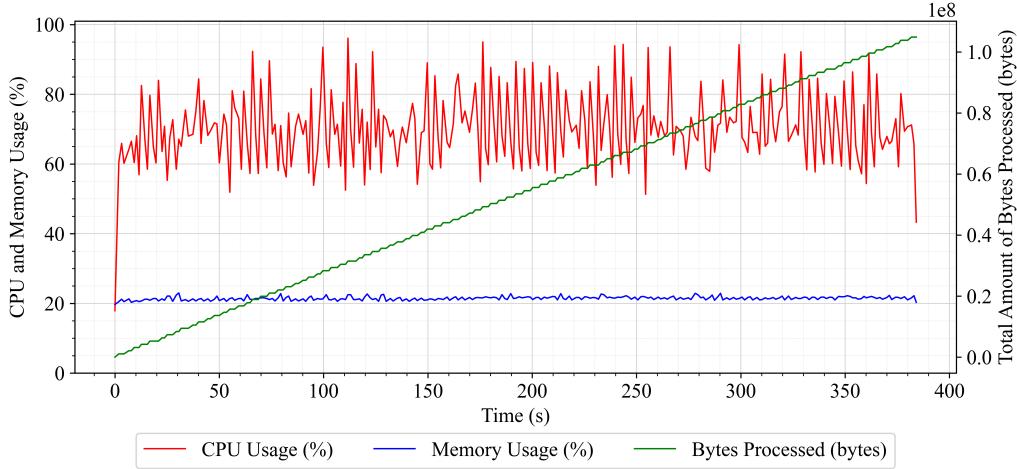


Figure A.6: Dual Core, Basic Test 2 Server and Client Metrics

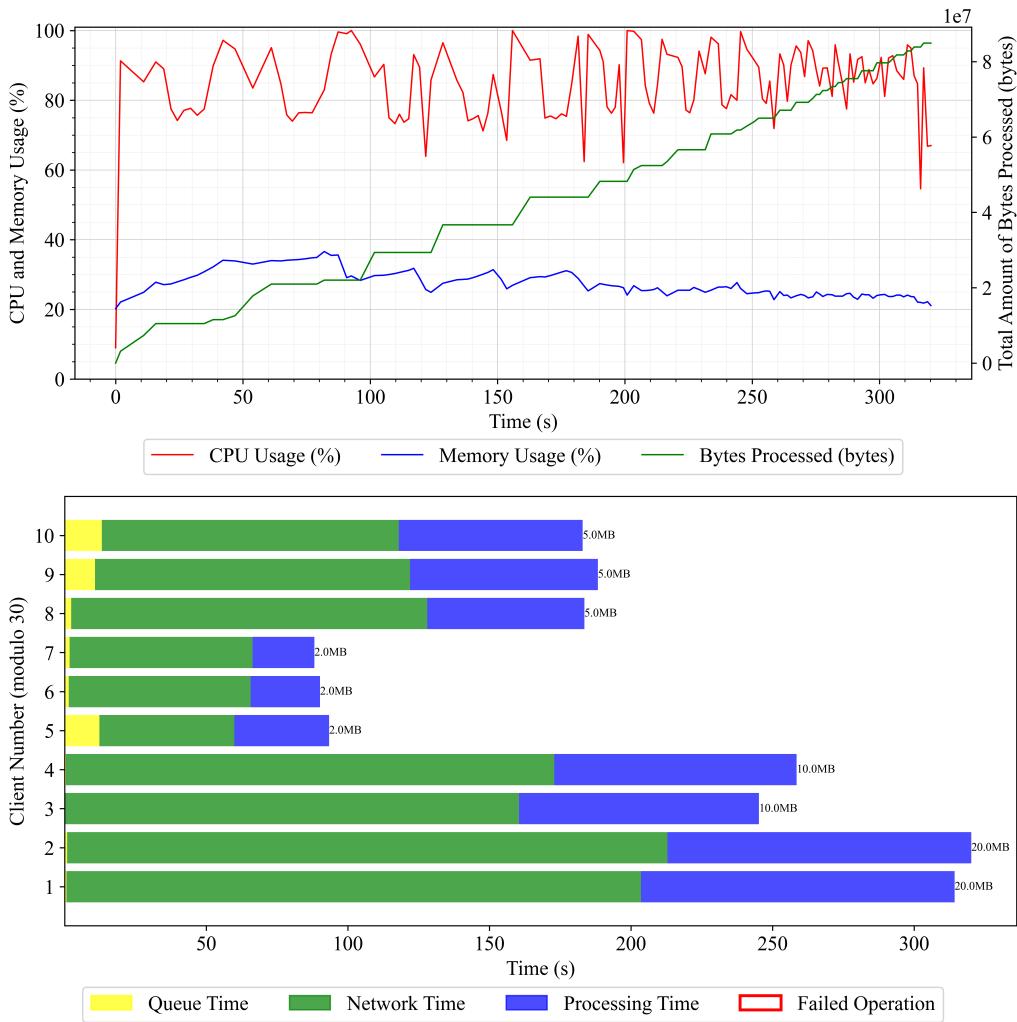


Figure A.7: Dual Core, Basic Test 3 Server and Client Metrics

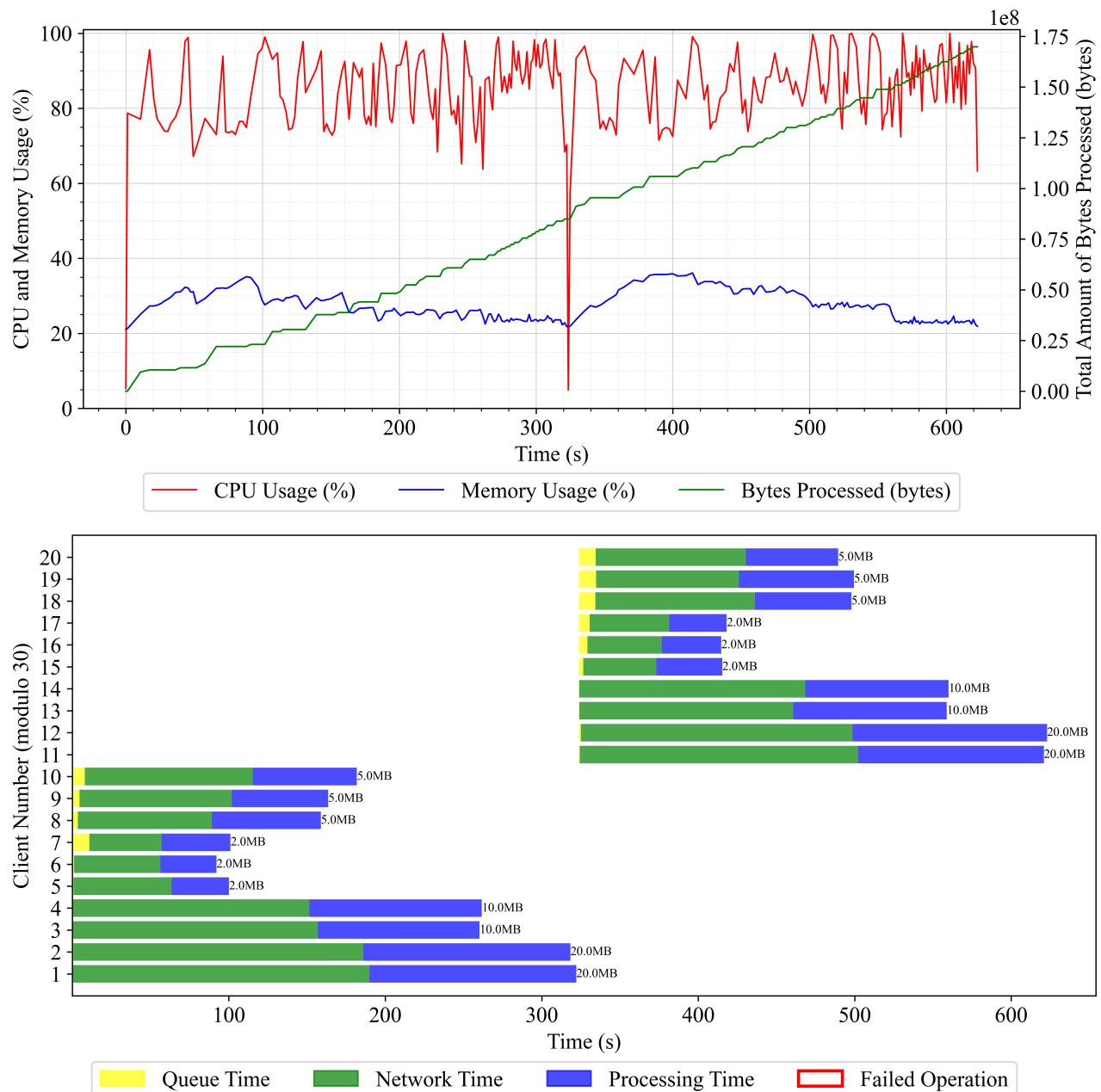
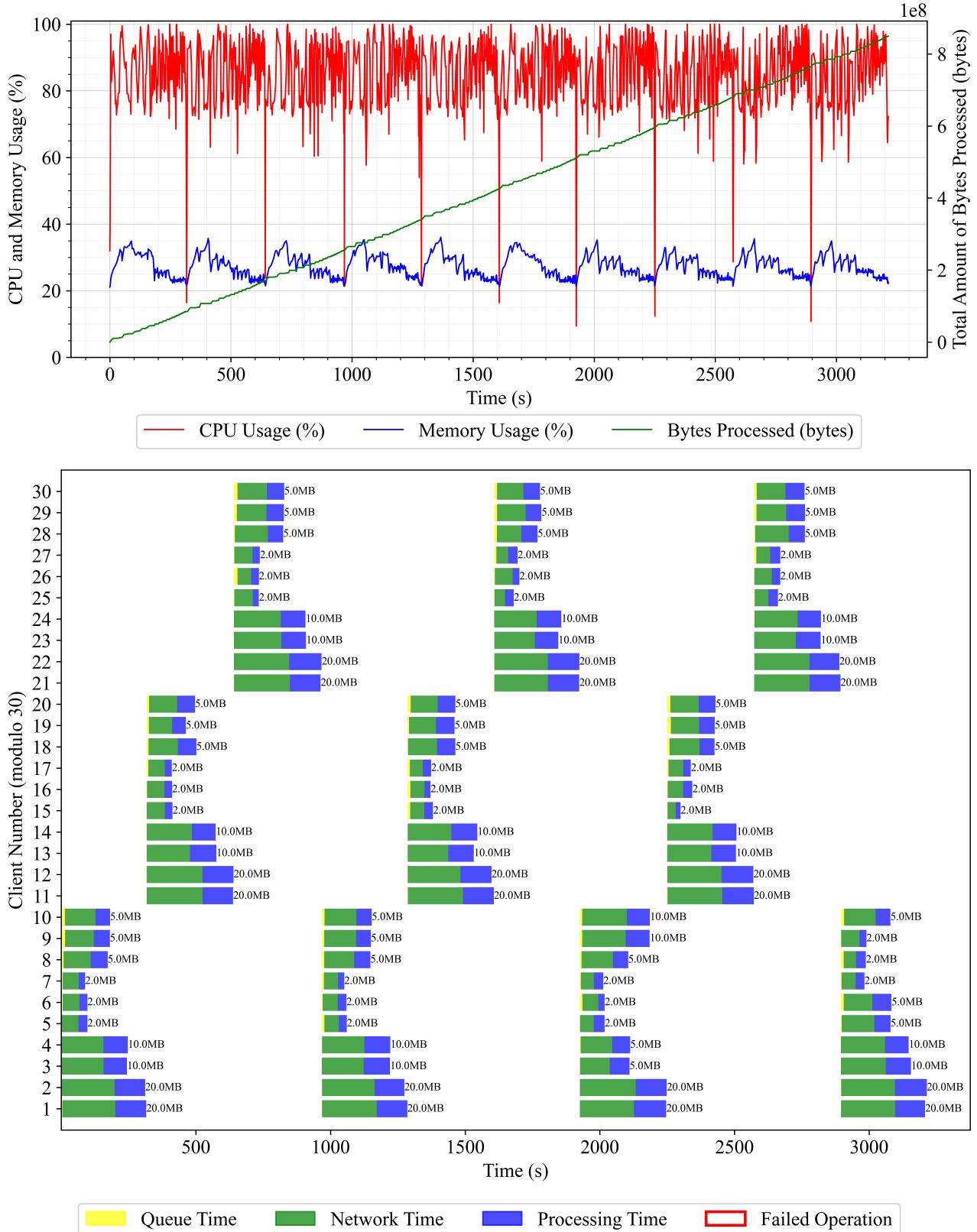


Figure A.8: Dual Core, Large Scale, 100 clients, Server and Client Metrics



A.3 Full Quad Core Results

Figure A.9: Quad Core, Basic Test 1 Server Metrics

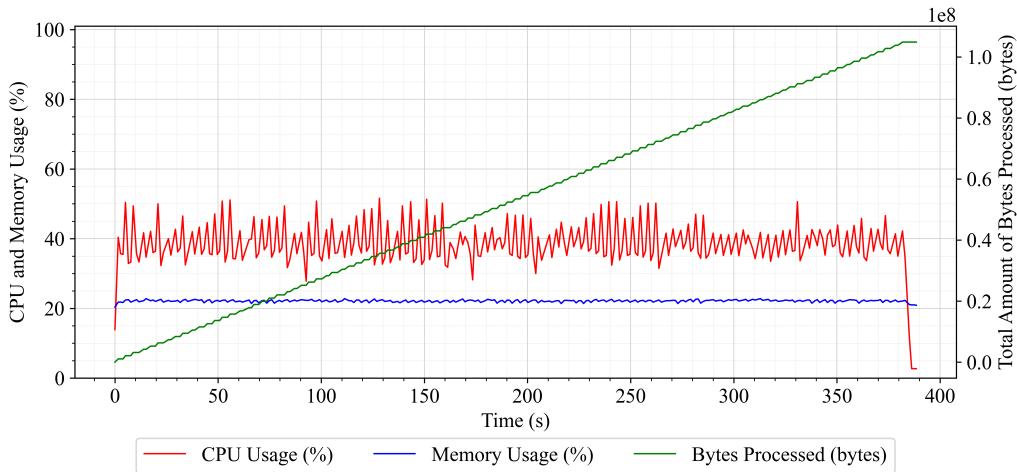


Figure A.10: Quad Core, Basic Test 2 Server and Client Metrics

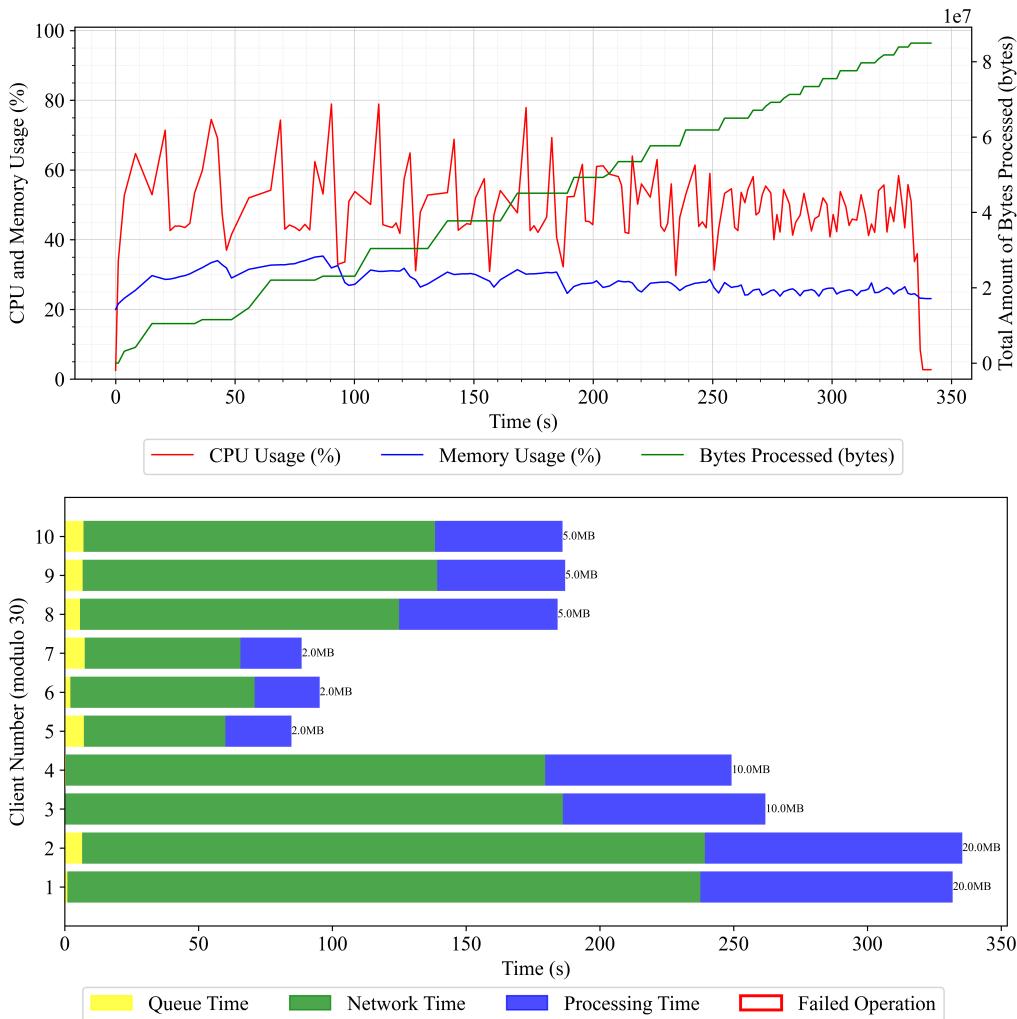


Figure A.11: Quad Core, Basic Test 3 Server and Client Metrics

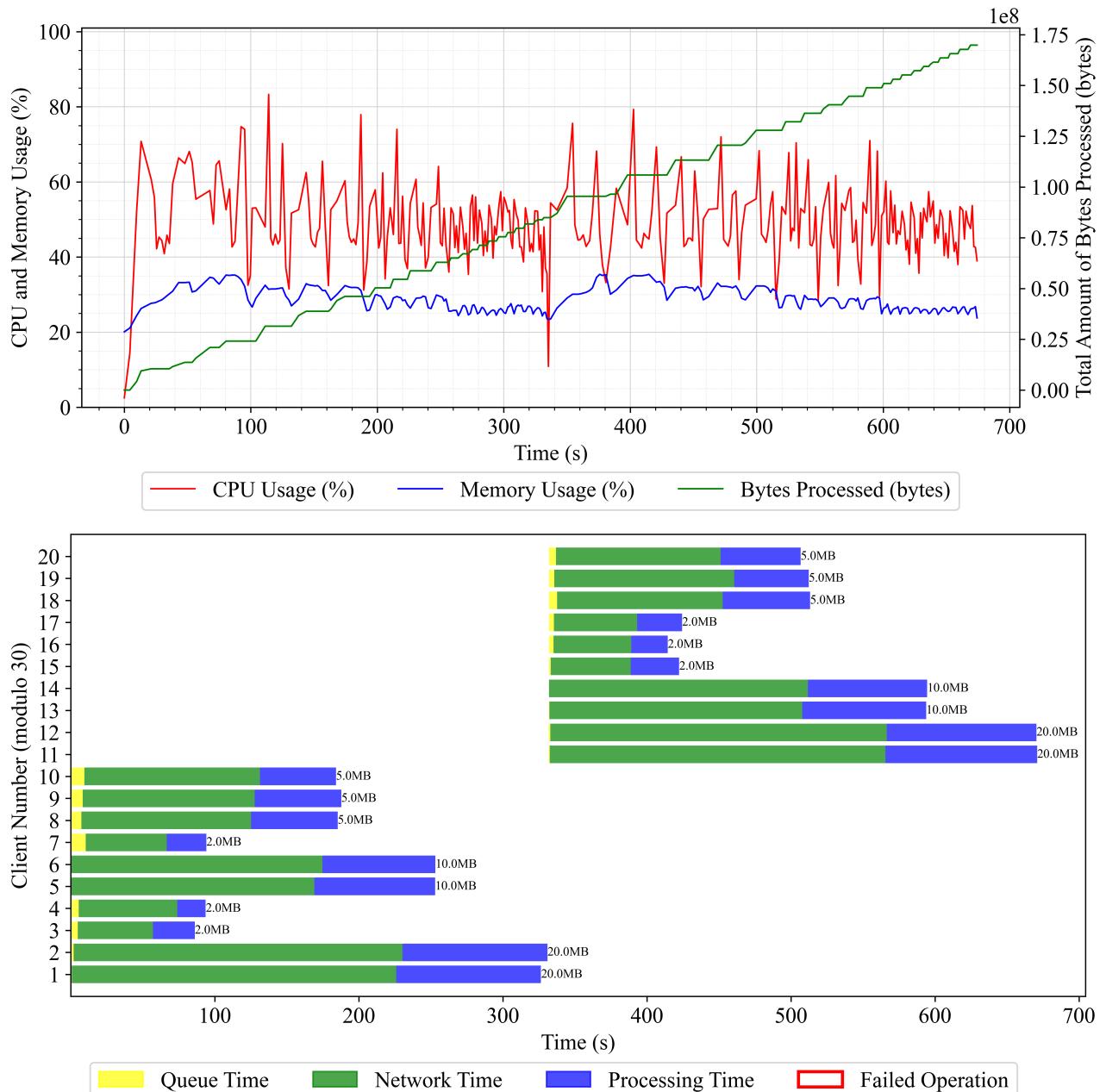
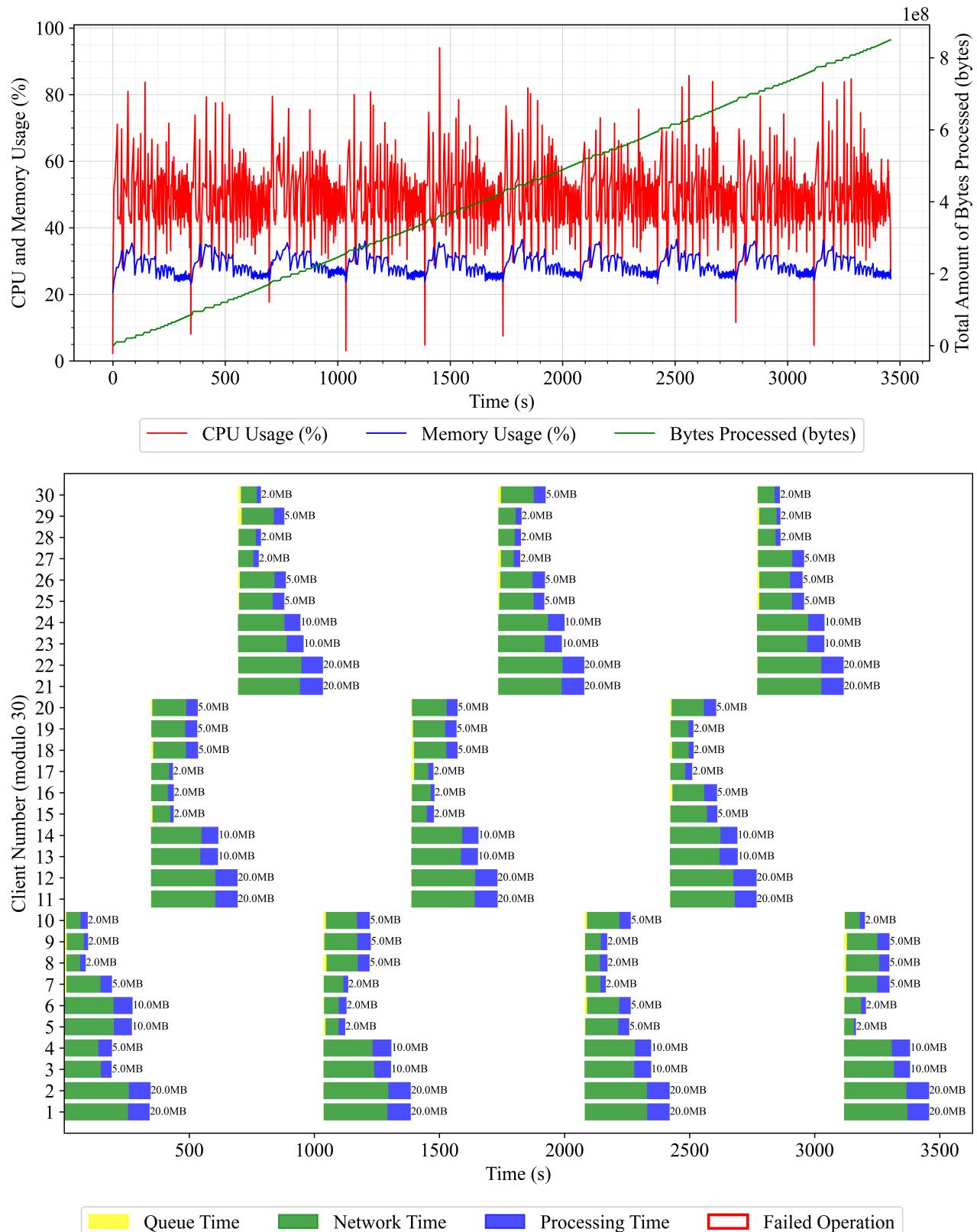


Figure A.12: Quad Core, Large Scale, 100 clients, Server and Client Metrics



A.4 Full Raspberry Pi Results

Figure A.13: Raspberry Pi, Basic Test 1 Server Metrics

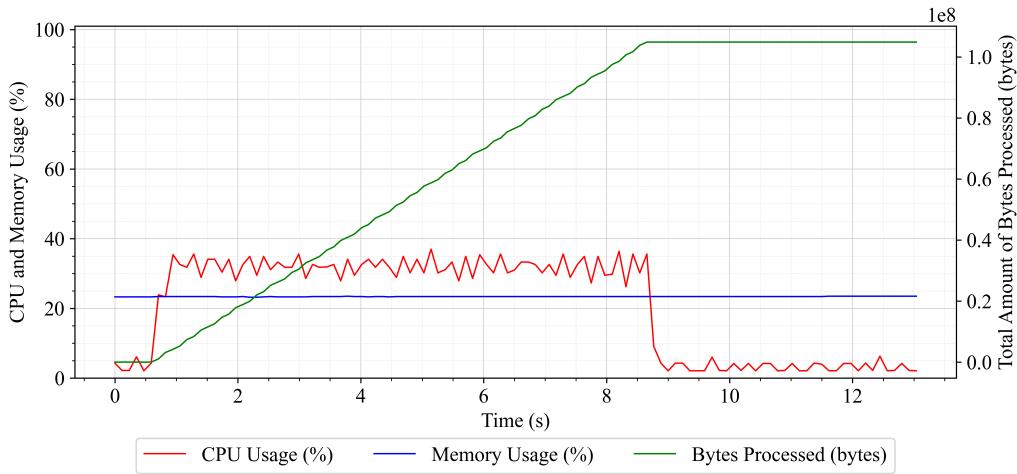


Figure A.14: Raspberry Pi, Basic Test 2 Server and Client Metrics

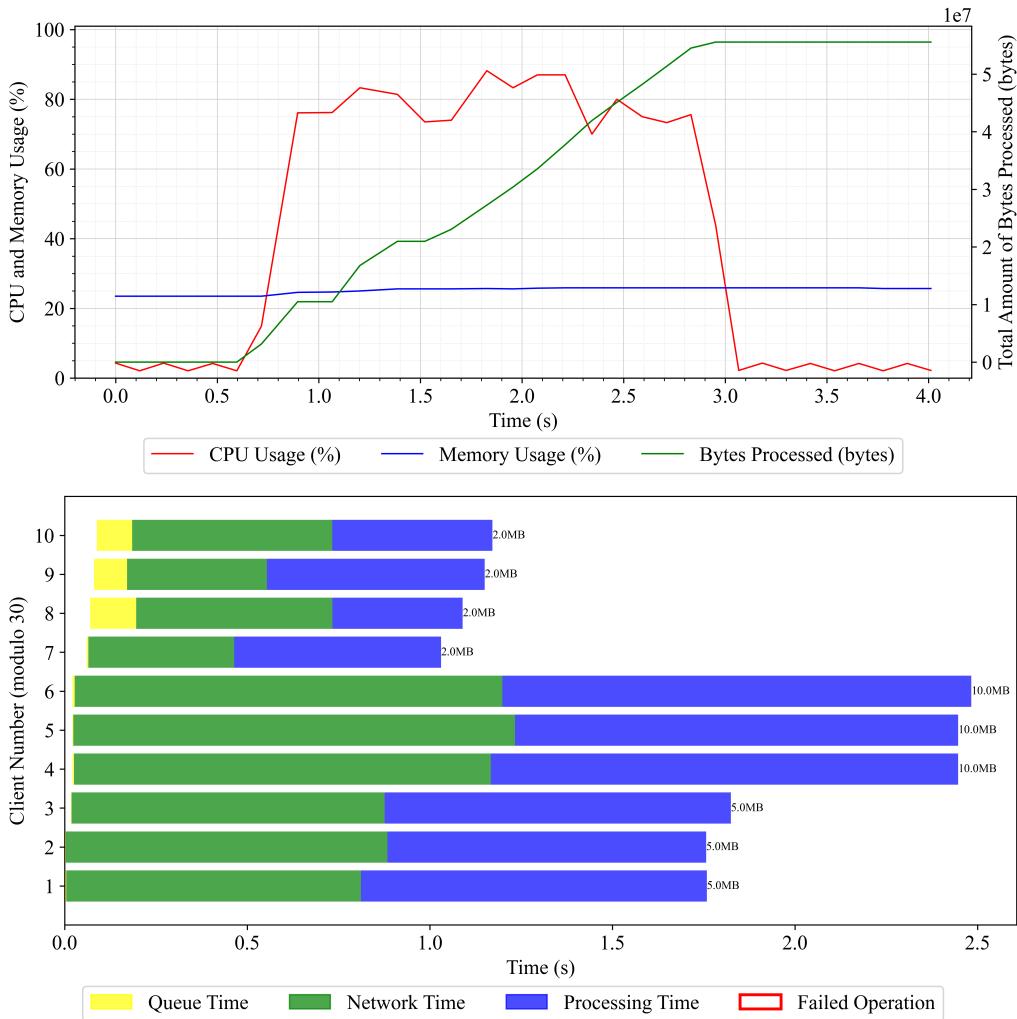


Figure A.15: Raspberry Pi, Basic Test 3 Server and Client Metrics

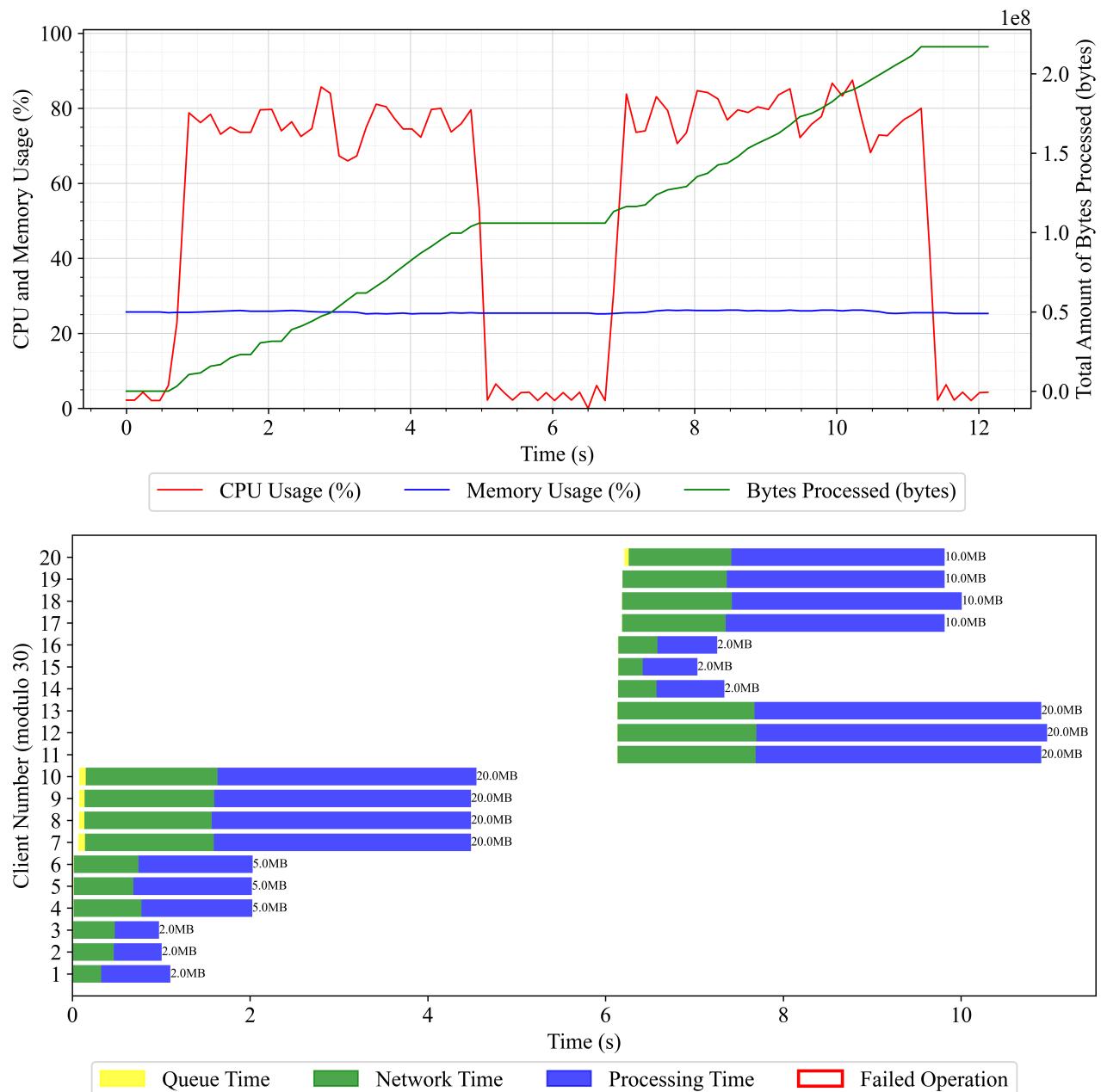
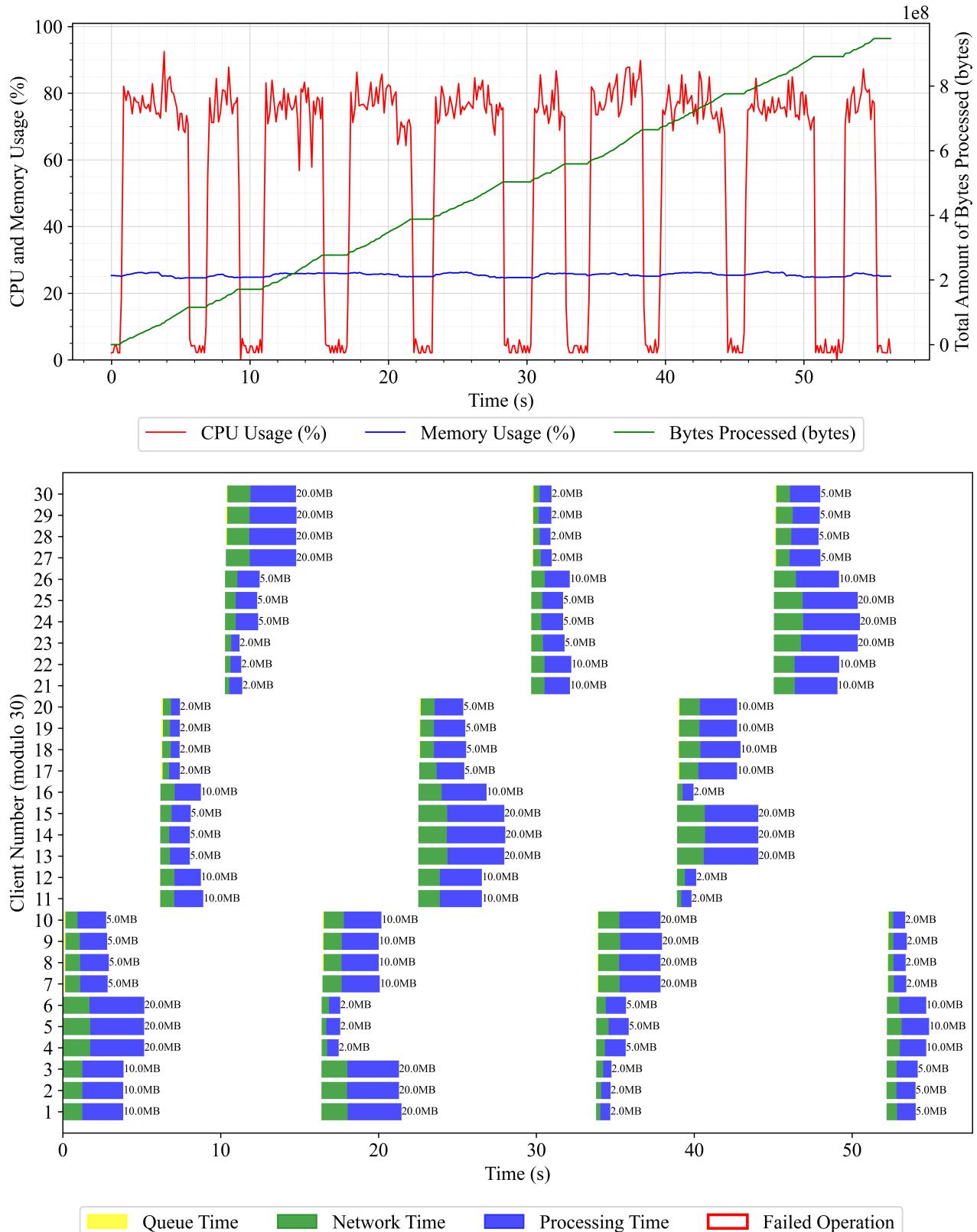


Figure A.16: Raspberry Pi, Large Scale, 100 clients, Server and Client Metrics



A.5 Full Improved Dual Core Results

Figure A.17: Improved Dual Core, Basic Test 1 Server Metrics

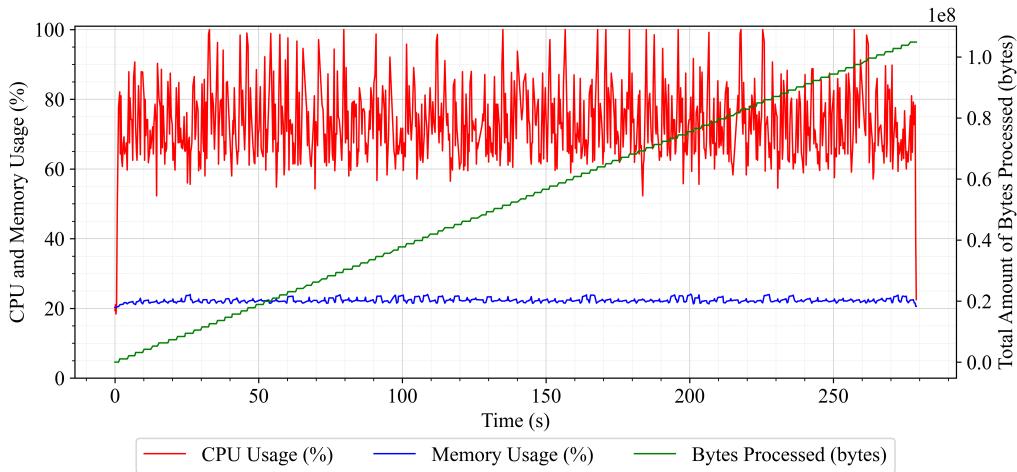


Figure A.18: Improved Dual Core, Basic Test 2 Server and Client Metrics

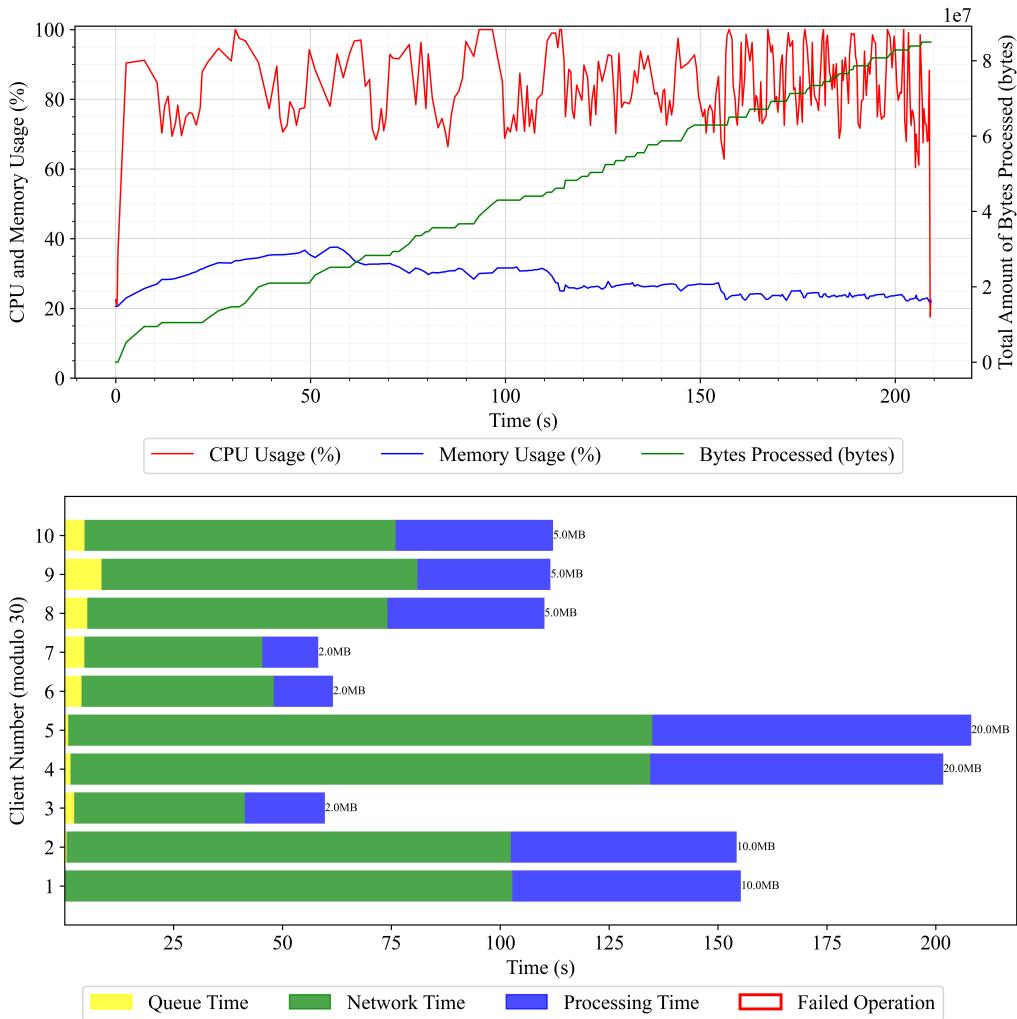


Figure A.19: Improved Dual Core, Basic Test 3 Server and Client Metrics

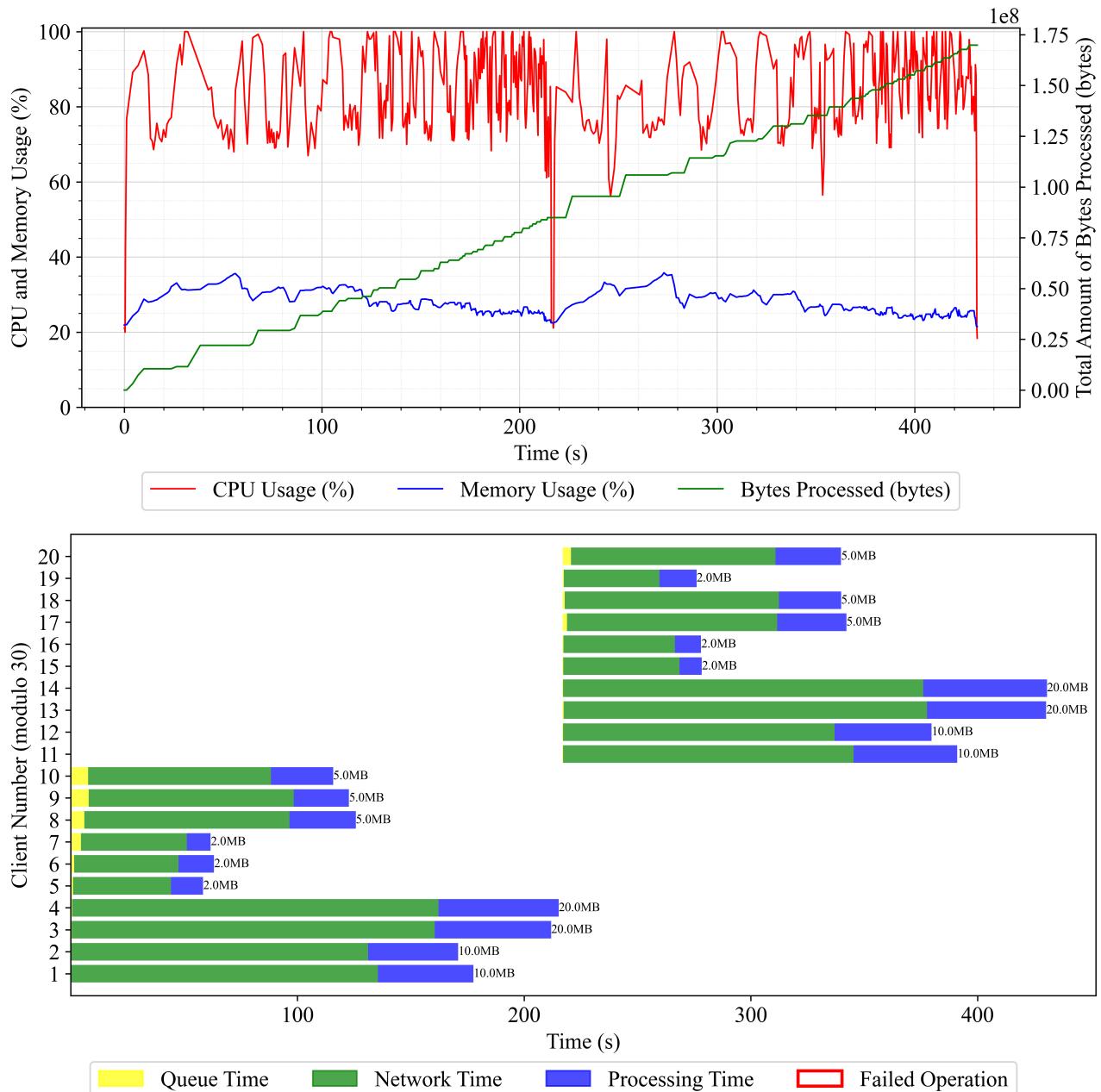
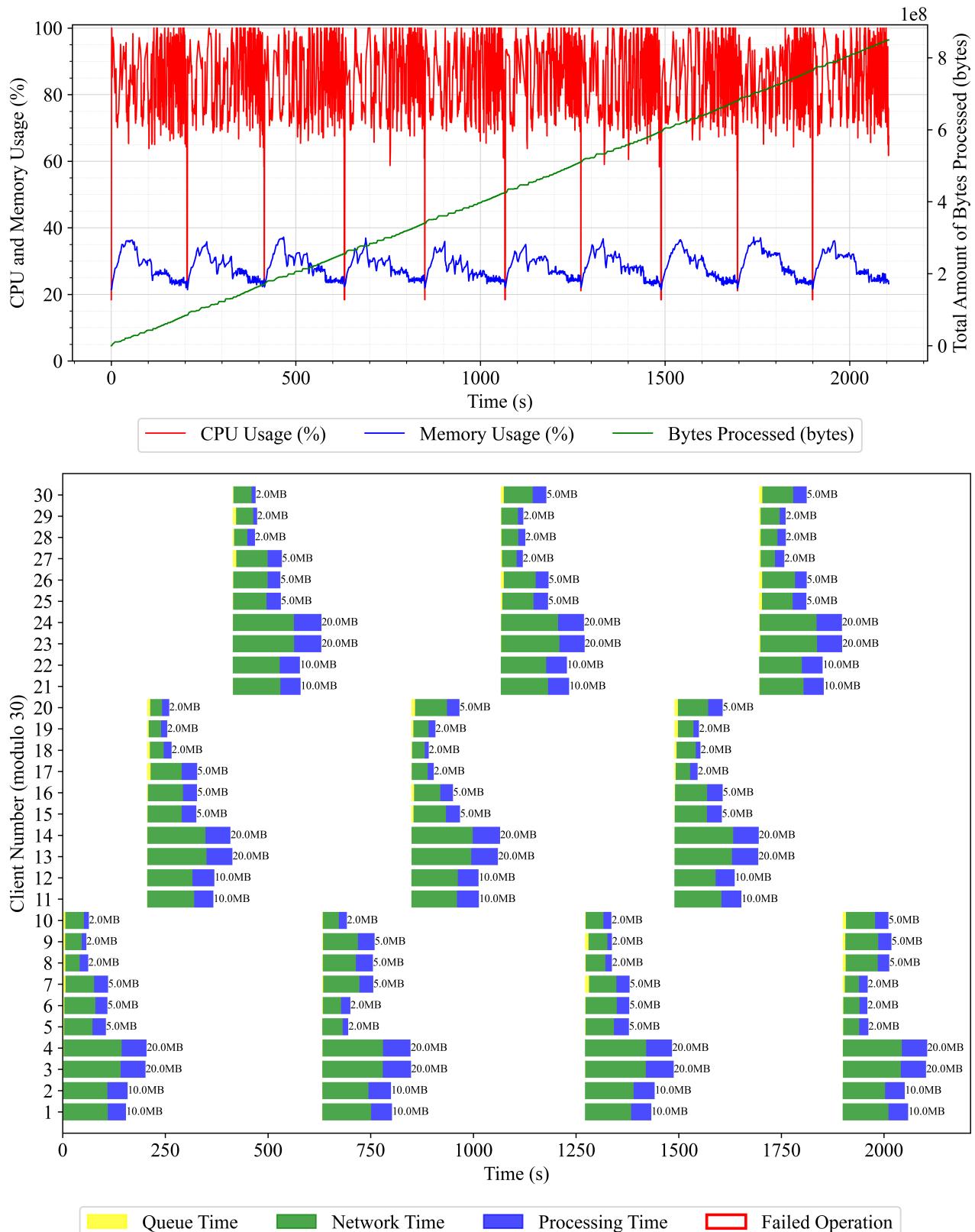


Figure A.20: Improved Dual Core, Large Scale, 100 clients, Server and Client Metrics



A.6 Name of Appendix-2

Endquote goes here.

Author of quote,
Source of quote