

High Performance Network Switch Solution Using Open Sourced Hardware

Zhouqiang Tu

MSc in Internet System and Security
The University of Bath
September 2016

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:

High Performance Network Switch Solution Using Open Sourced Hardware Accepting Dual Feeds

Submitted by: Zhouqiang Tu

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the author unless otherwise specified below, in accordance with the University of Bath's policy on intellectual property (see <http://www.bath.ac.uk/ordinances/22.pdf>).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

The thesis introduces an experimental dual-feed solution over open source hardware to reduce the latency in network. One significant improvement in the solution different from other established commercial solutions is that the network infrastructure is built upon open sourced hardware, and the main goal of the project is to prove that the cost in implementing ultra-low latency network can be reduced significantly by using the open sourced resources assisted with FPGA. The project focuses on implementing an experimental environment of fat-tree network topology over three Raspberry Pi2 Model B with one Spartan FPGA, with the implementation of MPICH2 among three ARM CPUs, providing scalability in computational scale to the system. The experiments performed in the project shows that the solution reduces network latency significantly with reasonable cost in bandwidth and computational power. The system can be implemented in both high-frequency market data infrastructure and high-performance computational clusters as the distributed switch, for it meets the requirements of: low-latency data transmission, reliable end-to-end communication and acceleration in collective communication in MPI.

Contents

1 Thesis Background and Literature Review	1
1.1 Thesis Inspiration	1
1.2 Literature review	2
1.2.1 Market Data Feed	2
1.2.2 Low Latency Network Infrastructure	2
1.2.3 A-B Data Feed	3
1.2.4 Scalable Interconnect Switches in Supercomputers	4
1.2.5 Field-programmable gate array(FPGA) and Integration with Open Sourced Hardware	8
1.2.6 Open Sourced Hardware	8
1.2.7 Similar Solution	8
1.3 Conclusion	11
2 Details of Implemented Technologies	12
2.1 Remote Direct Memory Access(RDMA)	13
2.1.1 Two Important Concepts: Verbs and Queue Pairs	14
2.1.2 Queue Pair	14
2.1.3 RNIC Verbs	15
2.1.4 Implementation Details of Verbs	16
2.2 Fat-tree topology implementation over RDMA	17
2.2.1 Reducing the Cost in Node Intercommunication	17
2.2.2 Analysing flat topology and fat-tree topology	18
2.2.3 Rebalancing the fat-tree over network bandwidth	20

2.3	RDMA verb implementation: Infiniband Verbs	21
2.3.1	Building the Passive Side	22
2.3.2	Conclusion	23
2.4	MPI-CH	24
2.4.1	Introduction to SPMD and MPMD	24
2.4.2	The architecture of MPI-CH	26
2.4.3	Conclusion	32
2.5	Field Programmable Field Array(FPGA) Assisted Computation	33
2.5.1	Sparta-6 integration with Raspberry Pi	33
2.5.2	FPGA hardware programming	36
2.5.3	Wishbone: interface for FPGA-to-PC communication	39
2.5.4	Conclusion	43
2.6	Conclusions for the Chapter	43
3	System Design and Implementation	44
3.1	Experiment Objectives	45
3.1.1	Overall Design	45
3.1.2	Main Objective	46
3.2	System Requirements	47
3.2.1	System Input: Dual feeds and UDP Session Simulation	47
3.3	Hardware design	53
3.3.1	Interconnection Design and Implementation	53
3.3.2	Power Supply Design	55
3.3.3	Conclusion	56
3.4	System Design and Implementation	56
3.4.1	Operating System and Internet Connection	57
3.4.2	MPI-CH2: Install and Configure	58
3.5	Software Design and Implementation	59
3.5.1	Design of the Data Feed	60
3.5.2	Dual-feed handlers and controller	63
3.6	FPGA Algorithm design	64

4 Experiment Design, Requirements and Expectations	65
4.1 Experiment Design	66
4.1.1 Experiment 1: Single data feed with single receiver	66
4.1.2 Experiment 2: Dual data feeds over the cluster	67
4.1.3 Experiment 3: Dual feeds processed with MPI processors without RDMA	67
4.2 Experiment Requirements and Expectations	69
4.2.1 Simulation of the data source	69
5 Implementation and Testing	70
6 Results	71
7 Conclusions	72
A Design Diagrams	78
B User Documentation	79
C Raw results output	80
D Code	81
D.1 File: yourCodeFile.java	82

List of Figures

1.1	Relationship among trade venue, data vendors and customers	2
1.2	Chicago-New Jersey microwave networks from data vendor Quincy Data . .	3
1.3	Overview of A-B streaming infrastructure	3
1.4	26 GPIO pins on the Logi Pi board	7
1.5	Beaglebone Black and Raspberry Pi2 Model B	8
1.6	Overview of A-B streaming infrastructure	9
1.7	Tianhe 1A interconnection network architecture	10
1.8	Benchmark result of LINPACK and HPL on Iridis Pi	11
2.1	RNIC Model Overview	14
2.2	Queued pair communication over internet	15
2.3	Flat structured network	18
2.4	Adding one node in flat mapping in the same field, six connections are changed	19
2.5	Adding nodes in tree structure, only two interconnections are changed . . .	20
2.6	Compression step k to step k+1	21
2.7	Passive side design	22
2.8	Active side design	23
2.9	MPI architecture	26
2.10	Rendezvous communication in RDMA	28
2.11	Structure of ADI3	30
2.12	The message queue between nodes	31
2.13	Fat tree implementation	32
2.14	Arduino MCU	34

2.15 Modern FPGA design	36
2.16 PWM modular logic	38
2.17 VHDL file structure	39
2.18 UART architecture	40
2.19 SPI schematic diagram	41
2.20 Wishbone register	41
3.1 System blueprint	45
3.2 Flow chart for filtering in the data feed receiver in BPMN	51
3.3 Interconnection hardware blueprint	54
3.4 Cabling on the switch board	55
3.5 Ethernet connection in slave node and master node	55
3.6 Power supply blueprint for the cluster	56
3.7 System blueprint	57
3.8 SSH terminal and VNC XWindows for master node	58
3.9 Setting up the MPI-CH2 environment	59
3.10 Data feed application blueprint	60
3.11 UDP sender from data loss rate to success rate	61
3.12 Dual sender flow chart	62
3.13 A-B data feed handler blueprint	64
4.1 Design of experiment 1	67
4.2 Design of experiment 2	68
4.3 Design of experiment 3	68

List of Tables

1.1	Latency and bandwidth in memory IO	4
2.1	Implemented Technologies	12
2.2	Average processing time for node N1	19
2.3	Wishbone functions	42
3.1	Content structure of system design and implementation	44
3.2	System Features	46
3.3	Low-latency market data handler metrics	48
3.4	Materials used in the experiment	53
4.1	Experiment design table	66

Chapter 1

Thesis Background and Literature Review

1.1 Thesis Inspiration

Financial instruments and organizations in the modern world needs to provide an live update on their current market status, and provide a package of information to its members and partners. The data they need to update is includes bid/ask prise, completed trades and other market status(Alexander, 2001). This package of information is aggregated, and streamed as 'Market Data Feed'. Modern financial market requires high performance infrastructure due to the implementation of automated trading system, which implements complicate market strategies and uses the data feed as sources(Le, 2009). Modern strategies requires huge computational power to perform mathematical modelling, and high performance network to make the order ahead of any competitors.

Low latency infrastructure is invented to meet the market requirements, it is usually consisted of two parts, computational nodes and network infrastructure. Existing solutions for the former part are customized servers, cloud computer farms and special data centres. They are programmed to perform data analysis over live market data feed, the outputs are trading behaviours packed as orders . Firms in computer network services are focusing on the latter part, as the latency in this part can be solved by engineering practises.

Solutions for low latency network usually come with high cost non-scalable hardware, and requires racks of server cabinets to install the devices. However, the entry barrier in the upfront investment for network infrastructure makes it difficult for medium or small financial institutions to start business in this field.

Researchers are looking forward to find alternative solutions to build network infrastructures that supports low latency market data feed. The researchers in high performance computation cluster provides a possible solution. The interconnect networks in supercomputers follows a different design in network interfaces. The improving design of parallel program architectural in supercomputers requires more efficient parallelism in network

interface(Pang, Xie, Zhang, Zheng, Wang, Dong and Suo, 2014). We introduce the high scalable design and the fat-tree topology in the architecture into our design of high performance network switches, and achieve similar acceleration in data process as the existing commercial solutions.

1.2 Literature review

1.2.1 Market Data Feed

The principle of providing low latency market data feed is to provide users live data of current price information and trader status, including individual portfolio valuation, financial market data, trading activities and alarms, a matrix of these data forms an 'Order Book'. The data feeds have different price for varies delays, famous market data feed providers, or the data vendors, such as Yahoo finance, ThomsonReuters and Bloomberg provide combinations of pricing and other market information in different latencies. For example, data delayed over 5 minutes can be fetched from Yahoo finance Application Programming Interface(API) for free(Yahoo!, n.d.), and ThomsonReuters provide paid services in different level of delay. Customers can choose to use designed data feed services to meet their business requirements, show in the reference to picture 1.1.

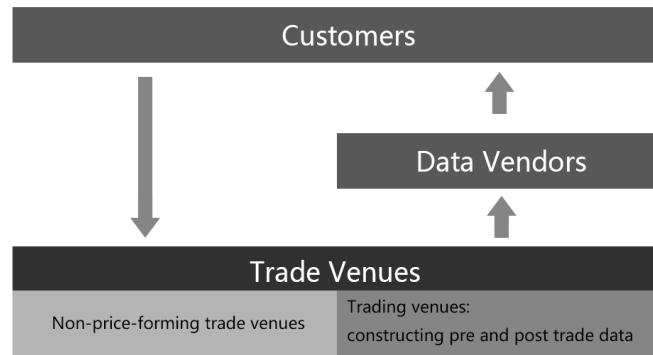


Figure 1.1: Relationship among trade venue, data vendors and customers

1.2.2 Low Latency Network Infrastructure

Engineers provide solutions to achieve lower latency in network transmission for market data, including direct redundant fibre cables from trading centre to traders, shortest path transmission and dual data feed. For example, solution using microwave towers reducing the cost of digging tunnels for fibres and enabling signals transmit over the sphere is im-

plemented between New Jersey and Chicago(Laumonier, n.d.), shows in the picture 1.2. Engineers also developed 'dual-feed' mechanism to reduce the cost in retransmission when package lost(Zusman, Tang, Nakelsky, Verbeck and Azizian, 1999).The introduction of dual feed, or the A-B feed of market data transmission uses two feeds with offset in delay to avoid significant failure such as delay or fault, especially for regional traders who want to trade across long geographical distances.

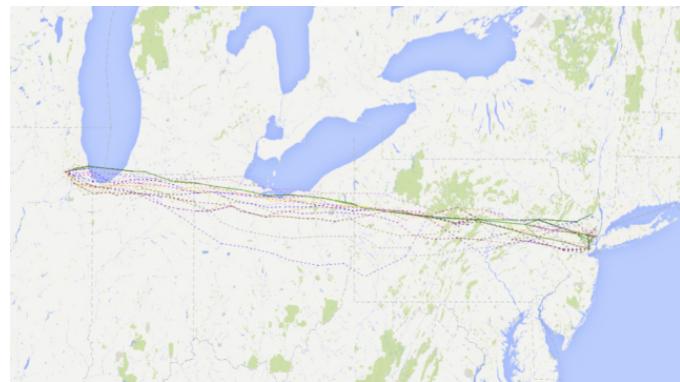


Figure 1.2: Chicago-New Jersey microwave networks from data vendor Quincy Data

1.2.3 A-B Data Feed

A-B feed structure requires two independent signals from one data provider, and the one feed(A) is several milliseconds faster than another feed(B). Relay stations between the sender and receiver need to decompress both feeds, filter and aggregate the package then transmit the data feed to the next station. Receiver also need a decompressor to handle the data from both feeds, reconstruct the original package and split the data into sub-streams forming process-specific subset of data, shows in picture 1.3.

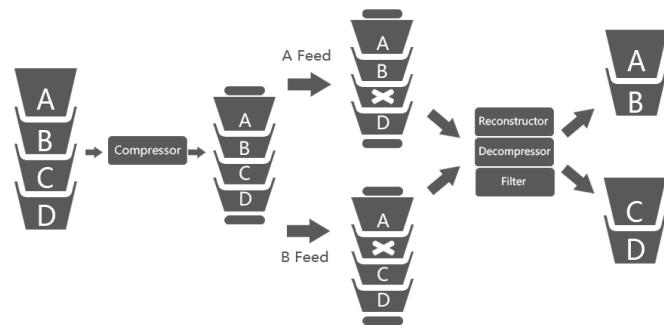


Figure 1.3: Overview of A-B streaming infrastructure

Each data feed arrives in sequence with a prefixed header, containing information from abstracted layers of network protocols(Postel, n.d.). UDP protocol suggests UDP client to provide reliability services that the receiver should perform CRC check of the received package chains, and retransmit the chain if package lost found, timeout mechanism exacerbate the difficulty in providing low latency data service. A-B data feed reduces enable the rapid collect-dispatch system to provide output within 500 milliseconds(Zusman et al., 1999).

1.2.4 Scalable Interconnect Switches in Supercomputers

Existing hardware solutions for low latency network is not suitable for mid sized institutions or individuals to start their usage of the network, and the development in building supercomputers provides an alternative solution to build infrastructure with equivalent function without implementing designed equipments.

Linear Speed-up in Beowulf Architecture

Beowulf is a highly scalable cluster architecture for computer clusters, and very popular in supercomputer design(Behrooz, 2005). The basic principle is providing scalable computational power by scaling number of cores and other resources, the nodes are interconnected in proper designed network topology. The bottleneck now lies in the communication among cores in paralleled program(Brown, 2004), the bandwidth and latency in interconnect network worsen the efforts in removing the obstacles in improving the performances of the Beowulf cluster.

Six kinds of memory are defined according to its logical location in computer, they have distinct latency in read/write and requirements in bandwidth, shows in table 1.1.

We set T_L is the total time cost for a parallel program running in a certain Beowulf

Table 1.1: Latency and bandwidth in memory IO

	Latency	Bandwidth Requirement
L1 cache	1-5ns(1 clock cycle)	-
L2 cache	4-10ns	400-1000MBps
Local memory	40-80ns	100-400MBps
Local disk	5-15ms	1-80MBps
NFS disk	5-20ms	0.5-70MBps
Network	5-50us	0.5-100MBps

cluster, and it is paralleled in n threads. Assume the ideal scenario that n is much smaller than the number of total computational cores N in the cluster, so the threads can run in separate cores without waiting. P_{L_i} is the possibility of finding code or data in the i th type of memory, t_i is the IO time cost for i th type of memory. We can have the expected total

time cost in equation 1.1:

$$E(T_L) = \sum_i^n T_{L_i} = \sum_i^n (\sum_j^6 P_{L_j} t_j) \quad (1.1)$$

The time cost is linear related to the increment of IO times in a single core, that rapid data or code IO in one core will increase the time cost significantly. Instructions are not loaded sequentially in L1 cache for most cases, so the bottleneck of accelerating Beowulf machines lies in reducing the swap among different memories. A simple solution is to break the memory requirements into small blocks, that the block is small enough to be swapped into L1 or L2 cache only once after execution, and runs without halt in content swapping until the result of calculation being transferred to the local memory. Master system dispatches the blocks to subsystems, and it is clear that the total time cost will be approaching to execution time of a single block when the number of subsystem increases.

Interconnect Networks Topology: Perfect Shuffle and Fat Tree

The Interconnect network in Beowulf clusters is important for acceleration and optimization. Performance of a supercomputer are measured by time cost in route permutations, and cost is defined by number of switching data(Leiserson, 1985). For a parallel computer interconnected by boolean hypercube, the physical volume cost for a n core cluster would be $n^{\frac{3}{2}}$.

The design of a layered interconnecting network called 'Shuffled Network' by Schwartz resolved the bottleneck in network volumes(Stone, 1971). The shuffle follows following rule, for a full set of N indices, each i maps into another set of indices by permutation P :

$$P(i) = 2i \quad 0 \leq i \leq \frac{N}{2} - 1 \quad (1.2)$$

$$P(i) = 2i + 1 - N \quad \frac{N}{2} \leq i \leq N - 1 \quad (1.3)$$

and the inverse permutation

$$P^{-1}(i) = \frac{n}{2} \quad (n \text{ is even}) \quad (1.4)$$

$$P^{-1}(i) = \frac{n-1}{2} + \frac{N}{2} \quad (\text{else}) \quad (1.5)$$

we introduce the binary representation of data in the original set, that $i = \beta_D\beta_{D-1}\dots\beta_1$, so the permutation of i is $P(i) = \beta_{D-1}\dots\beta_1\beta_D$, the inverse permutation of i is $P(i) = \beta_1\beta_D\beta_{D-1}\dots\beta_2$. This defines an ideal switch network that for o th step, the data can be divided into even-numbered and odd-numbered processors, and in the $(o+1)$ th step, data in even-numbered processors can be moved to low-numbered processors by implementing permutation P , and data move from odd-numbered to high-numbered processors can be

performed by using inverse permutation P^{-1} (Schwartz, 1980). The arbitrary movements of N elements in shuffle is $O(\log N)$, similar to the full hypercube network while decreasing significantly in network volume(Clos, 1953).

The implementation of a perfect shuffle interconnection is the Fat-tree network. For classic parallel programs, divide and conquer is an efficient solution(Aho and Hopcroft, 1974). Fat-tree interconnection strategy is a full binary tree. Message routing through children to the parent will carry data from all previous children, thus, the data capacity of the parent $\text{cap}(p) = \sum_{\text{child}}^{\text{route}} \text{cap}(\text{child})$. Data lost in during transmission from child_i to parent_i will be detected by the parent node, and retransmission will only occur between the two nodes(if the package still in the cache of the child node). The only exit is the root node, so any message route through the map will have guaranteed reliability with minimum cost.

Comparison: MPI supported network interconnection and SPI direct data transmission

Serial Peripheral Interface(SPI) is another commonly used interconnection technology especially between two single chip machines(Junger, Moritz, Lueders, Thuss, Elbracht, Haensel and Kostorz, 2007). It uses direct digital data feed trigger by the chip, and translated into pulse normally to the General-Purpose Input/Output pins, which is a general known hardware ports for pulse signals(Lin, Huang, Yuan and Bai, 2008). The SPI uses slave-master structure, and abstracted the data transmission process, especially the hardware details in digital-simulate signal translation into high level programming language functions and interfaces.

This can be another ideal way to form an interconnected network among the chips, however, we need to compare the SPI technology to the MPI supported Ethernet interconnection. First, the features of SPI data transmission: data in applications can be translated into digital signals and send over GPIO ports shown in picture 1.4. The transmission do not need to follow any network layered protocols, and the SPI works as a translator between two applications. The speed of bit exchange in SPI is much faster, and much more efficient than the Ethernet protocols. However, using SPI introduces the problem of scalability and expense: as discussed above, the GPIO pins can only transmit one bit on each pin, and we can either choose to send a series of data via one of the pins, or send multiple bits of data in parallel over multiple pins.

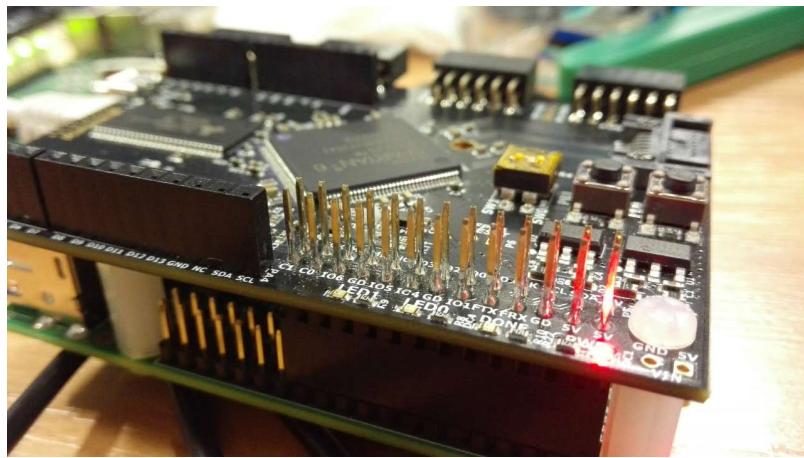


Figure 1.4: 26 GPIO pins on the Logi Pi board

The former solution slows down the system because even 1Kb data needs to be translated into 1,000 electric pulse with 50MHz system clock, using 20usecs. The other solution makes it hard to increase the scale, because nodes in this cluster needs to be able to broadcast, therefore the number of GPIO pins one node needs is $N_{node} \times n_{GPIO}$, N_{node} is the total amount of nodes, and n_{GPIO} is the number of nodes for a bi-directional interconnection between two nodes.

The MPI supported interconnection Ethernet network, however, do not need to worry about the two problems listed above. The defect of this solution is, it requires 10Gb Ethernet peripherals in the simulated switch cluster, and the equipments is hard to install and purchase.

We decided to use the MPI supported solution for the listing reasons:

- The SPI solution requires special hardware between each node for synchronization of system clock, and the solution is more focused on the hardware part
 - We cannot overcome the difficulty in implementing the GPIO port board, as the open-source hardware only supports 40 GPIO pins on the board, however, the minimum number of boards is three. Therefore the SPI interface can only send 20 bit each time between the nodes, and the experiment cannot perform the scalability test as we do not have expandable GPIO pin peripherals
 - We found a solution to simulate the scenarios in UDP datagram transmission in 10Gb Ethernet network, especially the package loss possibility simulation, over the normal RJ-45 Ethernet network. The solution will be illustrated in the following chapters.

1.2.5 Field-programmable gate array(FPGA) and Integration with Open Sourced Hardware

The introduction of Field-programmable gate array, or the FPGA, aims at solving the heterogeneous computation requirement in hardware based acceleration and general computing by using gateway logic circus to perform general algorithms or general processes(Hauck and DeHon, 2010). The FPGA is an integrated circuit, with re-configurable gateway and other logical blocks that can simulate most non-reconfigurable hardware with similar efficiency. The FPGA can be programmed to meet certain functional requirements by certain hardware definition languages, and modern FPGA also supports advances programming language interfaces, for example, C and Python.

The integration of FPGA and open sourced hardware have been carried by the community, and supported by the FPGA providers such as Xilinx and Spartan. The LOGI-Pi is the very first FPGA supported by the open sourced hardware, with Spartan FX9 FPGA chip and SATA connector directly to the designed hardware board(FX, n.d.).

1.2.6 Open Sourced Hardware

Hardware engineers need to design and print their customized motherboards to support FPGA and other peripherals, however, the industry produces general motherboards supporting most of the ports including PCI-E, HDMI, USB2.0 and USB3.0. Among which the most well-known commodity hardware is Raspberry Pi and BeagleBone Black showed in picture 1.5. Both are designed as System On a Chip architecture.

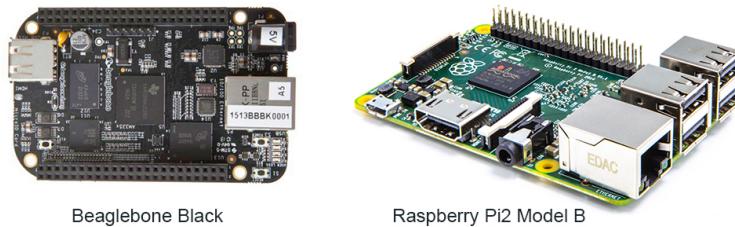


Figure 1.5: Beaglebone Black and Raspberry Pi2 Model B
(register .Inc, n.d.)

1.2.7 Similar Solution

FPGA accelerated market data feed

Engineers are not satisfied with the traditional CPU based acceleration solution, and developed an alternative hardware architecture that solves the problem of buffering and filtering high frequency data. This solution introduces direct connection between FPGA chip and

network interfaces, the Celoxica AMDC board enables dual feeds for single chip. Compressors and filtering program are burned in the FPGA(Morris, Thomas and Luk, 2009). The acceleration of stream processing in this solution lies in the customized hardware algorithm in FPGA, and data streams goes directly to the RAM of user cores via DMA message tunnel, the IO frequency of which is higher than that in network interfaces, the structure is shown in picture 1.6.

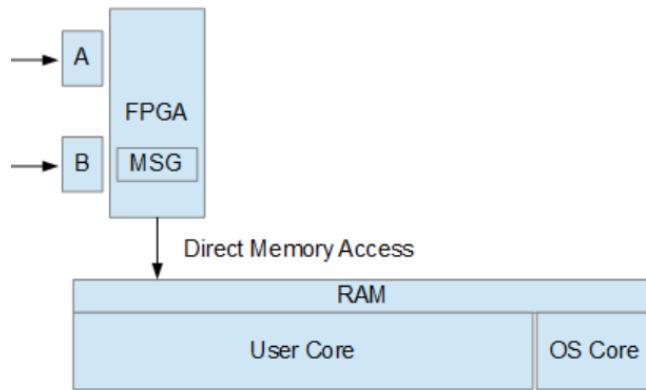


Figure 1.6: Overview of A-B streaming infrastructure

However, this solution requires high investments in hardware, especially the motherboard and printed FPGA. Scalability is another weakness of this solution, it does not provide easy solutions for large scale implementations, especially in the scenario of bandwidth upgrading from 1Gb to 10Gb Ethernet. The NIC chip and interfaces need to be redesigned and reimplemented in the printed FPGA chip, and against the principle of System On a Chip, that the algorithm and application should be maintained only by the system supported by the hardware, not the hardware itself(Klaas, 2004).

TH Express interconnection network

TH Express interconnection network is the local area net framework for the Chinese Galaxy series super computers. It is a fat-tree topology network, and consisted of two main parts: communication network and monitor/diagnose network(Yang, Liao, Lu, Hu, Song and Su, 2011). The former one connects computational nodes, the latter one checks machine status and resolves runtime error. The researchers developed a scalable network interface hardware High-Radix Routing Chip, and using RDMA method to access multiple memory spaces with unified offset, reducing processing time, improve the scalability and increasing the potential bandwidth. The architecture is shown in the picture 1.7

Solution provided by TH Express is efficient but too expensive to achieve, however, it provides an approach to reduce latency in high frequency network data supporting the massive

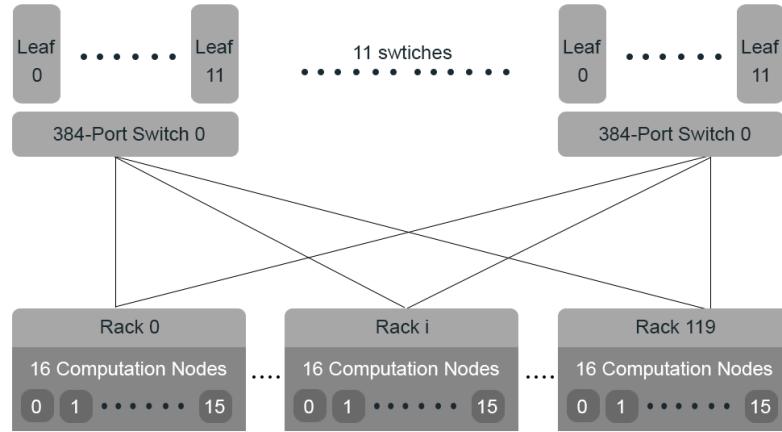


Figure 1.7: Tianhe 1A interconnection network architecture

parallel processing system, that scalable commodity hardware supported by RDMA can improve the reliability and bandwidth of the network.

Enyx's ARM-based System on a Chip solution to market data infrastructure

Market data hardware infrastructure providers offer variates of solutions to reduce the latency in market data feed. The most recent solution is to implementing FPGA hardware algorithm with ARM based SoC. It accepts A-B feed from normal network interfaces and perform the decompression and filtering via algorithm in FPGA, the ARM CPU focuses on mission dispatch and status monitor.

One of the well-known solution provider is the Enyx, the platform of which is based on Altera Stratix V FPGA board. The rebroadcasting latency for the FPGA supported market data processor can reach 1050 nano seconds, and average latency is 1300 nano seconds(Cisco, n.d.). Other commercial solutions supports implementation of trading logic in the switch FPGA, that the terminals from the traders only monitors the macro status of the system, and execution of trading signals only relies on the hardware implemented applications, which reduces the event-reaction time significantly.

Raspberry Pi supercomputer in Southampton University

The Southampton University has implemented an Raspberry Pi supercomputer with sixty-four Raspberry Pi Model B boards, interconnected several switches and named 'Iridis-Pi'(Cox, Cox, Boardman, Johnston, Scott and Obrien, 2014). The performance of the cluster is measured using LINPACK and HPL benchmark, and the single node performance in LINPACK and node increment benchmark by the HPL results are shown in the

picture 1.8. The cluster proves that the Raspberry Pi can form an usable parallel computing cluster, even though its power consumption is relatively very low compared to formal parallel computing infrastructures.

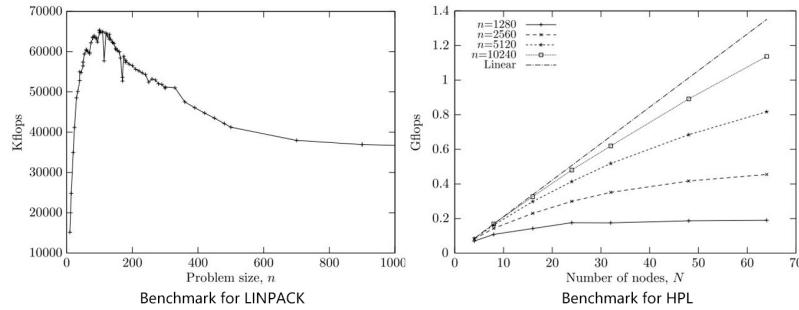


Figure 1.8: Benchmark result of LINPACK and HPL on Iridis Pi
(Cox et al., 2014)

1.3 Conclusion

In this chapter we made introductions to the background knowledge of high-frequency market data and its network infrastructure, especially the A-B data feed it uses to reduce network latency. Then we explained how the scalable interconnect switches are structured in the supercomputers, and focuses on the Beowulf architecture, which is the prototype of our design, the perfect shuffle within the interconnection network and Message Passing Interface. We made a comparison over the Serial Peripheral Interface connection and MPI supported network connection, and explained our choice of the MPI supported one. We also introduced the Field Programmable Gateway Array which holds our hardware algorithm, and two different types of supported open sourced hardware, the BeagleBone Black and Raspberry Pi 2Mode B.

We also made introductions to two established dual datafeed solutions for market data infrastructure and discussed pros and cons of both solutions. The high performance network switch in TH Express network brings us an new idea about the approach to the dual data feed hardware that a Beowulf structured open source hardware can achieve similar performance over the network with much less cost. The Raspberry Pi cluster in Southampton University proves that the performance of open sourced hardware can be acceptable if they are tuned correctly.

The following chapter will have detailed introduction of the technologies we implemented.

Chapter 2

Details of Implemented Technologies

This chapter discussed the details of the four key technology in the project: Remote direct memory access, fat-tree topology network, MPI-CH, and field-programmable gate array assisted CPU computation.

Table 2.1: Implemented Technologies

Technology	Implementation	Details
Remote Direct Memory Access	Infiniband RMDA	Infiniband verb implementation over RDMA data engine and protocols
Fat-tree topology netowork	Fat-tree over infiniband RDMA	Fat-tree network for RDMA internet layer to communicate more efficiently
Message Passing Interfaces	MPI-CH standard v2	Integrating the hardware platform to a multi-process cluster over a shared memory managed by RDMA
Field Programmable Gate Array	Sparta FPGA	Introducing FPGA to assist CPU clusters handling problems that are hard to be separated into paralleled threads

The implementation of these methods are carefully selected, to fit the architecture and achieve the best performance. MPI structure with RDMA is not a new technology, however, it has not been implemented over the dual-feed switches yet, so the design of the structure is different from a normal MPI cluster. The network topology in implementation

is also another innovation because this design is usually in the computational nodes, and the project introduces fat-tree RDMA network to reduce the requirement in hardware performances of single node.

2.1 Remote Direct Memory Access(RDMA)

The introduction of remote direct memory access technology is inspired by the architecture design in TH Express high performance interconnection network switches, which implements the RDMA as a basic memory space management infrastructure to enable the scalability of the system using distributed cluster as a virtual switch rather than using customized central switch. The basic principle of using RDMA is to trade for efficiency with the cost of memory space and bandwidth within the switch system.

Remote direct memory access technology is developed over the traditional Direct Memory Access engine, filling the requirement of encapsulating details in data copying and shifting in cloud platforms(Archer and Blocksom, 2012). The technology enables developers ignoring the implementation in data transmission among computation nodes, and focusing on the design of the parallel computing algorithms themselves. The actual RDMA protocol family is consisted of three protocols:

- Remote Direct Memory Protocol
- Direct Data Placement Protocol
- Maker UDP Aligned

The protocol family focuses on providing data availability over the interconnected networks without operating system kernel involved, and direct data exchange in the network interfaces. The technology requires no additional buffering spaces and performs under atomic operations: read/write, send/receive(Russell, 2012).

The problem of supporting remote direct memory access is the implementation of user-level data access over traditional network protocols, which would inevitably require the interception from system kernel model(Liu, Wu and Panda, 2004). Remote DMA requires interception before data swapping and access in the physical memory, therefore interception in kernel model must be implemented. However, the feature required by applications that RDMA should hide the details of data shifting and other system-level details, and application structures should not be infected by the implementation. For example, data shifting within the multi-processing distribution system should not infect the processes of single threads, which may require virtual data blocks whose actual address is on the remote node. Another common scenario is the asynchronous exiting sequence of different threads, which requires asynchronous data shifting in entering and existing actual memory spaces. Engineers have introduced several features in the implementation of RDMA protocols, that bypasses the kernel model in the traditional internet protocol layers, and created a virtual

tunnel among buffers in the threads running on separate nodes in the cluster.

RDMA plays as a core part in the project, which enables the open sourced hardware performs integrated computation with incoming data feeds and share without CPU involvement, minimizing the computational power cost by increasing the bandwidth and memory space cost, which is far easier and cheaper to achieve than high performance processors.

2.1.1 Two Important Concepts: Verbs and Queue Pairs

The implementation of verbs and queue pairs enables the RDMA performing kernel bypassing and CPU-free message handling. The verbs are abstract APIs that can be called in the application layer indicates that the application is implementing RDMA thus the layers below, especially the traditional socket-based internet protocol layers will be covered and the network interface card should be ready to be turned into remote memory direct access enabled model, which will handle UDP packages with DDPP header different from other packages that it will be processed by the NIC card and sent to local memory directly with RDM protocol management. The queue pairs can be viewed as the schedule handler that performing memory registry and calling the RDMA enabled NIC, or the host channel adapter(HCA) to caring pinning of the memory packages. The basic model overview is shown in picture 2.1

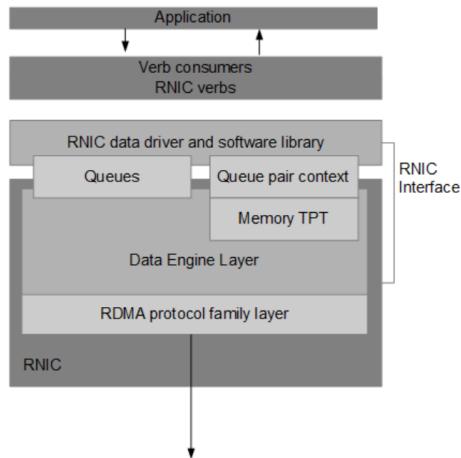


Figure 2.1: RNIC Model Overview

2.1.2 Queue Pair

The queue pair model is based on the consumer-provider model, which handles queuing of the tasks in an asynchronous way, and implements the actual data models in the network interface level to support the verb functions. The queue pair solves the problems of:

- Asynchronous arrival of data blocks in the interconnected network
- Fault detection in data transmission

The queue pair is an integrated system of software, hardware and framework in remote network interface card(RNIC), and the implementation of RMDA protocol family is in the RNIC framework. The protocol family can be divided into three levels: the *l2* level Ethernet access mechanism implementation; the *l3* level IP protocol layer implementation, including IPv4, IPv6 and IPSec; the *l4* level protocols for TOE over TCP protocols, and RMDA protocol family including RDMA, DDP and MPA(Arndt, Beukema, Craddock, Gregg, Schmidt and Walk, 2003). RNIC infrastructure enables TCP transport support for TCP packages bypassing system kernels, which intercepts data exchange from NIC to memory.

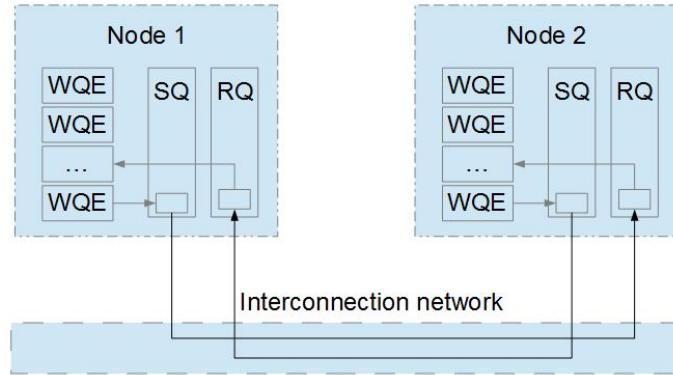


Figure 2.2: Queued pair communication over internet

2.1.3 RNIC Verbs

The RNIC verbs describes the behaviour of RNIC infrastructure, and creating programming interfaces for applications to make use of the infrastructures(Hilland, Culley, Pinkerton and Recio, 2003). The RDMA is built upon a consumer-provider model, and the verb methods act as the middle ware between the application consumers and actual queuing pairs inside the RNIC software logics.

The verb specifications defines functions and semantics the application need to access the RDMA protocol layers, however, the verbs only include connection management and tear-down semantics, and availability to additional protocol layers are not defined in the verbs.

2.1.4 Implementation Details of Verbs

The implementation of RNIC verbs are related to the actual policy defined by the queue pair politics, and different in actual infrastructure environment. In this project, the RDMA is the memory accessing system which provides buffering services for dynamic hardware structure for high performance system.

Memory management

RNIC introduces two concepts in memory management: memory window and queued pairs. The memory window manages the organization of contents in each node, and queued pairs defines the way separate nodes communicating with each other.

The memory windows are the basic swap element between the actual memory block and RNIC buffer(Garcia, Hilland, Culley, Barron and Krause, 2006). The RDMA introduces memory windows mechanism to perform memory registration and management. The remote memory shifting and offset table management are implemented through memory windows.

Memory space management are performed through RNIC data drivers via Tagged Offset(Boyd, Joseph, Ko and Recio, 2007). From the application consumer view, the memory spaces are continuous and the application can visit the entire 'shared' space by applying simple offset, but it works different from the queue pair view.

The queue pair is the mechanism which work requests are translated by the verbs into the interfaces provided by RNIC library, and be ready to enter the implementation of the data engine before going into the actual internet protocols. The working queues are maintained by separate nodes, and perform as the basic element of content swapping instructions. The swapping of the memory content, instructed by working queue I/O, is managed by STag access control:

Listing 2.1: STag access control

```
swapContentViaRNIC( QueuedProcessID qpid ){
    MemoryPointer mwPointer = getLocalMemoryWindowByQPID( qpid ) ;
    MemoryPointer mwDestiPointer =
        getDestiMemoryWindowByQPID( qpid ) ;
    //Get the destination from queued pair
    ReceiveQueue rq = getReceiveQueueID( qpid ) ;
    try{
        //Pop qpid from the send queue
        SendWorkingQueue.pop() ;
        SwapMemory( mwPointer , mwDestiPointer ) ;
    }catch( WorkingQueueNullException e ){
        e.printStackTrace() ;
    }catch( MemoryException e ){
        e.printStackTrace() ;
    }
}
```

```
//Push the qpid into the receive queue
re.push(qpid);
}
```

So the problem lies in the strategy in choosing the strategy in the cross-node STag management. We can choose either using the same queued process id for all queued pairs, and client can seamlessly access the process registered with the same id in the server, or using different process id on each queued pairs, and managed by the client-server communication. Both ways have problem: for the first solution, multiple clients can access each other's process for they enjoy the same id in the server, which is public accessible; the second solution cannot fulfil the requirement of unified memory window management in multi-client environment, which servers need register multiple memory windows for separate clients.

The solution provided by RNIC is a combination of both solution:

- At binding phase: queued pair id binds with each memory window, and different in separate nodes;
- At accessing phase: memory window with QPID is registered in both source and destination paired queues with same QPID;

2.2 Fat-tree topology implementation over RDMA

2.2.1 Reducing the Cost in Node Intercommunication

Cost in node intercommunication lies in two aspects: the content transmission speed between the network interfaces and memory spaces, and the latency in data links among nodes. Data swapping within RDMA system is a bandwidth consuming performance, and the efficiency in data transformation is the threshold for the overall efficiency of the system. In the data engine level, NIC can access the data blocks with the direct memory addresses without CPU, and improving the accessing speed of data by direct I/O performances from NIC to memory.

Listing 2.2: Direct Data access from NIC to memory

```
SwapMemory(Memorypointer source , Memorypointer destination){
    //The pid in different client is different
    QueuedProcessID destiPID = receiveQueue.pop().getPID ;
    Memorypointer desti =
        destination + getOffset(source , destination) ;
    try{
        kernel.directWriteToMemory(desti , destiPID.getContent()) ;
    }catch(MemoryIOException e){
        e.printStackTrace() ;
    }
}
```

In the network topology level, the RDMA network protocols introduces fat-tree topology, which 'parent' nodes have the bandwidth aggregating bandwidth of all children nodes, and the perfect shifting mechanism guarantees the minimum cost in data shifting.

The implementation of perfect shifting is based on one assumption: most of instructions performed by applications require data from adjacent memory blocks $T_{i+1}(d) = Addr_i + V(T_i(d))$, therefore, the 'next step' data requirement process always requiring data either:

- Require data from local memory space
- Require data from the neighbour nodes with the same parents

The implementation of dual feeds in data sources also reduces the time cost significantly, for the bypassing of kernel model in UDP package processing, the client in the network interface in switches works different in error processing, and node level error can be detected in leaf level switch. Comparing to flat structured centralized switch, the fat-tree topology interconnection reduces the expected error detection time from $E(T_t|e) = E(T_{node}|e) + E(T_{route})$, however, the expected time of processing error in node($T_{(node)}$) is a constant, assuming the cluster is homogeneous, and reducing the error processing time in network transmission phase.

2.2.2 Analysing flat topology and fat-tree topology

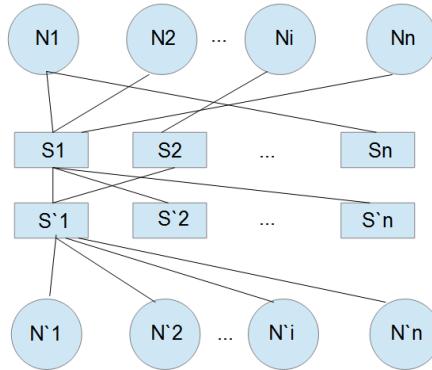


Figure 2.3: Flat structured network

The nodes can be divided into three groups according to the topology position to each other:

- Adjacent nodes: nodes have the same parent node
- Non-adjacent nodes in the same field: nodes have different parent nodes, but the nodes have direct interconnection

- Non-adjacent nodes in different field: nodes have different parent nodes, and the nodes have no direct interconnection

As shown in picture 2.3, the N_i nodes can detect and process errors in transmission and calculation, and S_j works as the network switches which cannot process data errors. The average processing time from node N_1 to adjacent nodes in network can be divided into three parts:

Table 2.2: Average processing time for node N_1

Equation	Comment
$E(T_{adj}) = T_{N_1 \widehat{S}_1 N_i} = \frac{\sum_i^n T_i}{n}$	i is the adjacent node
$E(T_{N_1 \widehat{S}_1 \widehat{S}'_1 N_j}) = 2 \times E(T_{adj}) + 2 \times T_{\widehat{S}_1 \widehat{S}'_i}$	j is the non-adjacent node in different field
$E(T_{N_1 \widehat{S} N'_k}) = 2 \times E(T_{adj}) + E(T_{\widehat{S}_1 \widehat{S}'_i})$	k is the non-adjacent node in the same field

From the equations it is easy to observe that the most time-consume process is the error detection when data is route through different fields, however, this could be frequent even we assume that the instructions in most cases process data in the adjacent node, because the switch nodes in different nodes in the node cluster, for example, N_1 and N'_1 could be logically adjacent in the computational cluster it connects to.

There are two options to reduce data processing time in the switch cluster shown in the table 2.2, one is to make all nodes adjacent to each other, however, the structure of the centralized hub could be over-complicated, another one is to connect all the fields together, the problem is the complexity in constructing the interconnection network. A full mapping among the field hubs requires each hub supporting bandwidth of $bandwidth_n \times n_{fields}$, which can be very costing if we need to implement the high bandwidth in each field node. One possible solution to avoid the increment of bandwidth requirement is to implement layered interconnection network rather than a flat structure, which resolve the bandwidth requirement to the back bone layer rather than the entire communication network.

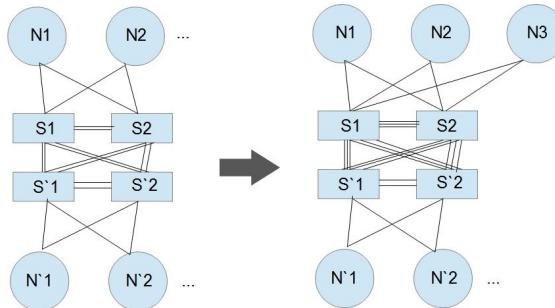


Figure 2.4: Adding one node in flat mapping in the same field, six connections are changed

The implementation of a tree-structure properly solves the problem, which not only provides a layered infrastructure which implements an aggregating bandwidth requirement from leaves to the root upstream, but also ensures the scalability of the system at the minimum cost, and the binary tree search time is $\log(N)$ (Ellis, 1980), because the increment in leaves only affects the sub-nodes in certain fields, rather than the entire system.

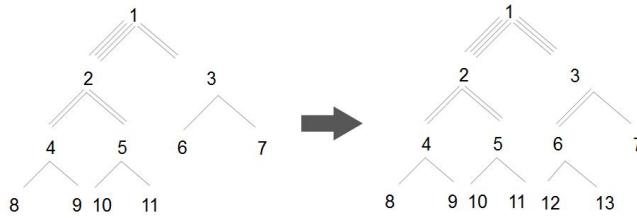


Figure 2.5: Adding nodes in tree structure, only two interconnections are changed

2.2.3 Rebalancing the fat-tree over network bandwidth

Another benefit of introducing the tree topology is the quick rebalancing feature of a tree topology network. The unbalanced tree will cause an unbalanced bandwidth load in the system. Advanced algorithm can rebalance the tree from a vine to a balanced binary tree with the time complexity of $O(n)$, and the amount of the nodes only have linear affect on the rebalancing practises. The extreme situation is a m depth vine, whose root only have one leaf in one side, and another vine with $m - 1$ depth vine.

First, introducing the compressing of a vine in the rebalancing:

```
Listing 2.3: Direct Data access from NIC to memory
compression(NodePtr root , Integer count){
    NodePtr scanner;
    NodePtr child;
    Integer i ;
    scanner = root;
    for i from 1 to count{
        child = scanner.right;
        scanner.right = child.right;
        child.right = scanner.left;
        scanner.left = child;
    }
}
```

and the steps from k th compression to $(k + 1)$ th compression can be shown as the figure below:

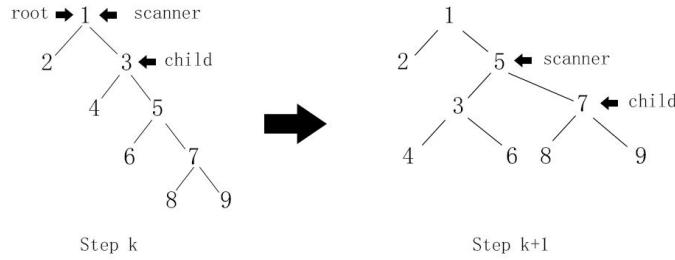


Figure 2.6: Compression step k to step k+1

The compression process will be performed each turn in rebalancing, for an vine with n depth, each step will make a compression of the rest $\frac{n}{2}$ branches, and the node of the vine can be a complete binary tree, take the example showed in picture 2.6, nodes 2,4,6,8,9 are all complete binary trees, and the compression will not change the structure of the binary threes, because the only the vine nodes 1,3,5,7 are changed by the compression process(Stout and Warren, 1986).

The rebalancing of the tree structure resolves the balancing problem of interconnection bandwidth, take the bandwidth requirement of node 5 for example: in step k before rebalancing, the bandwidth requirement of node 5 on the left side is b_1 for only one node, and on the right side is $\sum_m b_i + 1$, m is the amount of the leaves after vine node 5. After rebalancing, the bandwidth of node 5 on each side is the same($2 \times B_b$, B_b is the bandwidth of a balanced tree).

The introduction of rebalancing tree in the network topology of fat tree is for the stable scalability, and new nodes adding to the system will not have to be added following certain topology rules: the balancing of bandwidth requirement are maintained by the master node automatically, and the topology of the system is maintained dynamically.

2.3 RDMA verb implementation: Infiniband Verbs

Infiniband verbs, or the iverbs, are the verbs supported by infiniband RDMA protocol family which supports the implementation of RDMA over general hardware clusters. The build-in network topology management can be configured into fat-tree, which is discussed above that we want to introduce in the system.

The Infiniband provides upper-layer protocols(ULPs) supporting not only RDMA itself, but also most features in MPIs. The reason not using the entire Infiniband protocol family is that the support of message passing interfaces in infiniband is ill-supported in scalability, and the version we use for multi-processor interface for the open-source hardware cannot perform ideally in the infiniband pre-configuration, because the passing interfaces designed in the iverb is for much heavier use, including distributed service providers and computation. Our system is a pure message passing system which only filtering and passing message for the clusters, rather than perform the computation for them.

The iverb implementation can be divided into two parts(Bedeir, 2010). The client part and the server part. Server side mainly listens to the sockets, and react to the sockets send from the client side. The client send sockets to the servers in order to fetch data from memory directly via RDMA mechanism.

The iverbs lie in the client side when the application need to communicate with the server(the details are hidden by the verbs) to fetch data and instructions, and the server side when request are queued in the memory spaces, verbs will translate the queues to working queue elements and find the memory windows it points to, reply with the correct memory space addresses to the client to fetch the data.

Two features must be guaranteed for the full functioning of iverb system:

- Asynchronous reliable communication: both sides will receive hanging notifications and instructions
- No buffers in application level: the synchronization of the send-receive system requires no buffering especially in application memory spaces, which will cause serious problems if the content of application data is modified after sending without completing.

2.3.1 Building the Passive Side

The passive side creates and maintains an event handler for queue pairs and an event listener running in the background of network interfaces, which listens to the events from the active sides and send request queues to the queue pairs in the queue event handler directly without implementation of CPU. The iverbs in the event handler will then send the response including the memory content to lower-level protocols to be packaged into sockets, and the network interfaces will then send the package to the destinations.

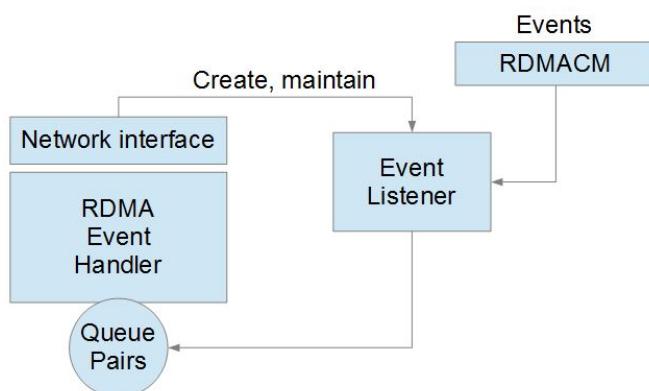


Figure 2.7: Passive side design

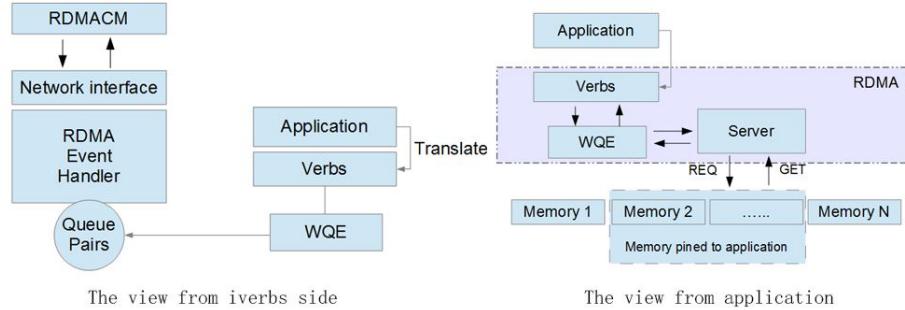


Figure 2.8: Active side design

Building the Active Side

The active side runs in the client, which directly queries the content from the 'virtual' memory spaces, which is managed by the server. The active side runs a queued event handler which consumes working queue elements translated by the iverbs from the applications, and resolves receiving queues from the server, getting the actual content of the shared memory spaces.

The active side maintains a route to peer nodes for both active and passive sides, in order to communicate among the clusters for swapping data and instructions.

2.3.2 Conclusion

This section introduces the RDMA method for memory management, and the fat-tree topology we need to implemented assisting the RDMA for better performances.

The purpose of implementing RDMA in the project is to overcome the weakness in computational power in open-source hardware, which is usually based on ARM processors with limited memory spaces. The established commercial solution to this problem usually contains an unified memory flash. The RDMA enables us to introduce fragmented memory hardware into an virtually unified one.

The purpose of implementing fat-tree topology is inspired by the TH-Express high performance computation clusters, which uses fat-tree interconnection network to solve the problem of low-latency network infrastructure(Pang et al., 2014). In this particular project, fragmented nodes makes it different than implementing a centralized routing system, which have fixed bandwidth balance. The implementation of rebalancing load among node clusters makes it possible to dynamically add/remove nodes.

2.4 MPI-CH

Researchers introduce the MPI to build a portable and scalable multi-threading platform over heterogeneous hardware environment(Dongarra, Otto, Snir and Walker, 1995). To understand the necessity of this interface, a brief introduction of message passing model need to be introduced.

Most parallel applications were for scientific purposes(Kendall, n.d.). Most of the libraries for such applications uses a message passing model for exchanging data among different cores of the machine, and normally a master manages multiple slaves by dispatching tasks and gather results.

The MPI-CH is an implementation of the message passing interface standard, and is widely used over multi-processing machines as a data exchange model over multiple processors performing multi-threading computations. It covers almost every aspect of MPI standards, and widely supports far more different hardware architectures compared to another well-known implementation, the OpenMPI.

The MPI-CH is the solution for implementing scalable dual-feed receiver in this project, to enhance the efficiency of communication among the nodes using a message passing interface, instead of building a complicate interconnection structure from ground zero. The project makes use of the listing features from MPI-CH compare to other solutions:

- Easy to build and configure message passing interfaces
- Remote memory access mechanism to enable the scalability
- Fat-tree topology to balance the efficiency in hardware architecture of single nodes and network bandwidth

2.4.1 Introduction to SPMD and MPMD

A parallel application can be divided into two different kinds according to the way it uses data and instructions, Single Program Multiple Data and Multiple Program Multiple Data. Before introducing the SPMD and MPMD,it is necessary to make a brief introduction to the concept of Single Instruction Multiple Data(SIMD) and Multiple Instruction Multiple Data(MIMD).

SIMD is the model that each node execute same machine instructions over different data, for example, for array A , executing $A_i + 1$ on each element of A can be performed by a SIMD process by implementing the *plus1* method on each paralleled cores. MIMD works in a different way: for calculation of equation $A + B + C - D + E \times F$, an MIMD implementation can divide the calculation into $(A + B)_{node1} + (C - D)_{node2} + (E \times F)_{node3}$ to three nodes with different instructions. Both SPMD and MPMD are the subsets of MIMD(Darema, 2001). In this project, MPI-CH library supports both SPMD and MPMD mode, the system can differ the model by using the command line tool indicating how the program and data being seprated.

Single Program Multiple Data

Single Program Multiple Data(SPMD) is the parallel application which only implements one application, and execute it among each node with different sets of data.

A good example of message passing model is a shellsort process: for an l length array A_l , the shellsort process divides the array into $\frac{l}{mod(\frac{l}{2})}$ fragments, and perform quicksort process on each fragment. The quick way to improve the efficiency of the algorithm is to place each quicksort process on a independent computational core, therefore, the time complexity only depends on the strategy of choosing the length of steps $mod(\frac{l}{2})$. However, it leaves a problem of communication among the cores.

For step k , the algorithm uses n amount of cores, and in step $k + 1$, the content in the memory space of each core will have the following two steps:

- 1. Merging together into a new array A'
- 2. Dividing A' into $\frac{n}{2}$ fragments and dispatching them to $\frac{n}{2}$ cores

Therefore, the programmer only need to design the dispatching mechanism and quicksort algorithm in each core. However, the solution could be cumbersome to achieve without MPI model due to the fact that for each step, the number of cores need to wake and perform send/receive practises will be different, and the designer need to predict the number of cores in each step. However, the time cost in communication is very high as the centralized node need to communicate to each node two times each step.

Multiple Program Multiple Data

Multiple Program Multiple Data performs parallel practises in a different way. Take the shellsort for example, if the parallel application is running on a heterogeneous cluster, whose node has different software and hardware structure and the programmer need to design and implement different sorting algorithm on each node. Therefore the step k to $k + 1$ could be:

- 1. Merging the fragments into new array A'
- 2. Querying the infrastructure library and forming the dispatching strategy for step $k + 1$, including the number of nodes, structure of the implemented nodes and data requirements for each node
- 3. Implementing $k + 1$ step, and divide the array A' into required fragments, transmit the data into the destination

The MPMD program is hard to design due to the difficulty in managing the data fragmentation process, which differs according to the configuration of each node.

MPI-CH implements both SPMD and MPMD models, and the host can differ the structure by using command line to submit tasks from master to slave nodes.

2.4.2 The architecture of MPI-CH

The core part of MPI implementation is the send-receive pair between processes, forming a programming interface following certain message protocols with semantic specifications over varieties of machine implementations.

The architecture of MPI-CH standard can be abstracted as the picture below:

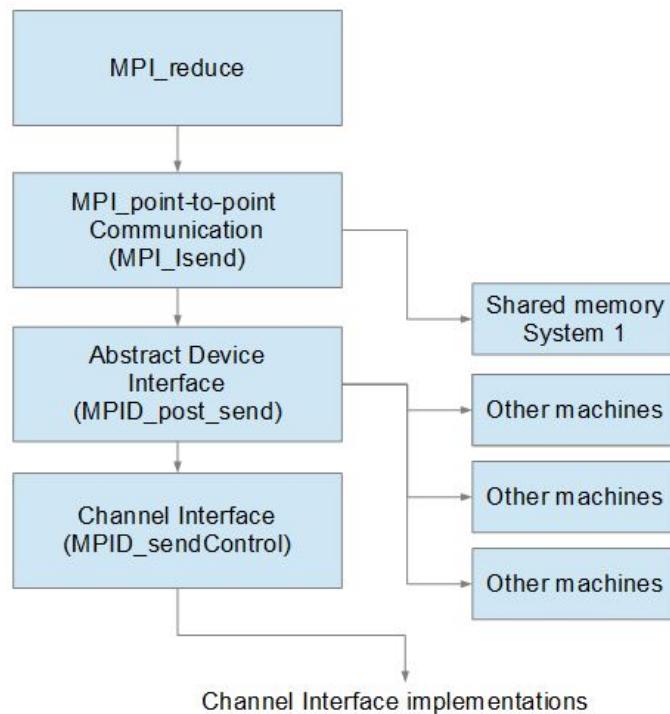


Figure 2.9: MPI architecture

The design for the MPI-CH infrastructure follows the following principles(Gropp and Lusk, 1996)

- Sharing code without compromising in performance over the actual implementation
- The MPI-CH should be easy to dock on different platforms with minimized amount of code changing

The first principle is to meet the need of re-implementation lower-level objects in MPI implementations over opaque objects, including communicators with tags on it, because

most of these contents in MPI-CH is platform independent, and code sharing will solve the problem of dynamic scalability. The second principle requires modularised design of the infrastructure, which can replace the platform independent modules, especially the sharing part, easily and dock to the platform with minimum cost.

The implementation of MPICH can be split into three levels according to the sequence from application to the actual hardware.

MPI API

The MPI reduce, or the MPI API for the applications, are the highest layer which implements user-level abstract objects and the user applications can introduce these elements to get access to MPI-CH managed resources.

The high portability feature of MPI-CH makes the interfaces easy to use and independent to actual hardware. One application can run on different MPI-CH supported hardware clusters by using configuration parameters. The entire application can be wrapped by two functions: MPI_Init(num_arg,*vec_args) and MPI_Finalize(). The program can perform both SPMD and MPMD inside the curve, and the MPI standard does not define the behaviour inside body(Org, n.d.a).

Listing 2.4: Example of using MPICH initialization-finalization pair in C

```
#include<mpi.h>
int main(int argc, char** argv){
    MPI_Init(NULL, NULL);
    int processor_size;
    MPI_Comm_Size(MPI_COMM_WORLD, processor_size);
    int processor_rank;
    MPI_Comm_Rank(MPI_COMM_RANK, processor_rank);
    char processor_name [MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    /*—————*/
    /* Perform algorithm */
    /*—————*/
    MPI_Finalize();
}
```

Communication mode in MPI

It is essential to introduce the communication mode defined in the MPI standard before moving to the actual implementation. The MPI standard defines four communication mode: *Standard*, *Synchronous*, *Buffered* and *Ready* modes(Dimitrov and Skjellum, 1999). It is easy to understand that RDMA always need to transfer small amount of data in high

frequency over the network, therefore the protocol strategy cannot be too complex. The RDMA introduces rendezvous communication protocol to perform the transmission.

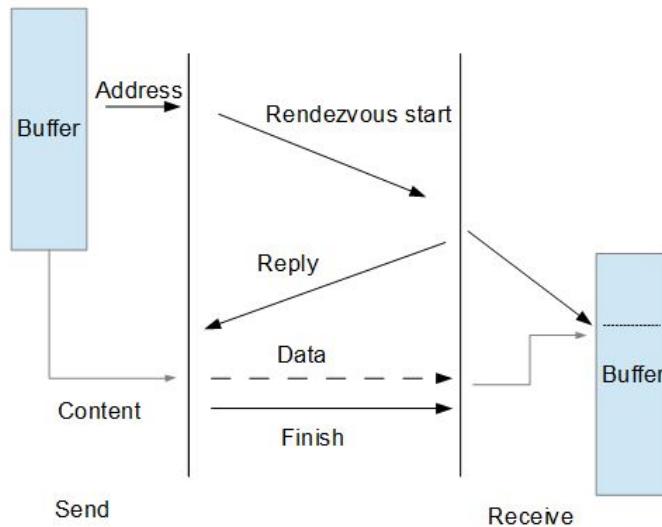


Figure 2.10: Rendezvous communication in RDMA

This zero-copy rendezvous protocol in RDMA works in the following sequence:

1. The buffer address is wrapped into control messages and transmitted to the receiver
2. The receiver decodes control message and find the address in the receiver's buffer, and send acknowledge message to the sender
3. The sender sends data to receiver, and send finish segment when the transmission is finished

It uses eager strategy in pushing data to the receiver, however, there are still problems in this send-receiver strategy, especially when MPI implements the RDMA feature. The problem will be discussed afterwards.

Abstract Device Interface

The MPI-CH implements the mechanism Abstract Device Interface(ADI) in order to provide abstract services for the upper level functions, and hide the details of hardware implementations to the applications(Gropp and Lusk, 1994). The message passing happens in this layer when the application uses MPI_send or MPI_receive, which indexed by the *handle*, and queued in a way to ensure the execution sequence. The queue is managed by the MPI-CH directly and abstracted from the hardware implementation, and the handlers

of IO behaviours are invoked by the lower layer, which slurp the contents and package them into proper format to suit the requirement of different mediums. The descriptions in handlers defines the type of interfaces it needs to be invoked.

The idea for the ADI design is to abstract the communications among nodes, and parallel scheduled from the view of applications should have no difference between local cores and remote ones. This due to the fact that MPI standard defines only the local performances in the first version of standard, and the development in network technology enables the hardware designer to introduce remote resources into a bundled parallel computing structure(Liu et al., 2004). ADI contains three queues: *send_queue*, *posted_recv* and *unexpected_recv*. Send queue is for the outgoing messages, and the other two are for receiving messages. The reason for dual queues for receiving is for the asynchronous communication. Application can have the two scenarios when receiving the message from other nodes in MPICH:

- The application is ready for receiving a message, and send *MPI_recv* request to the ADI layer via the API
- The application is not ready for any new messages, however the ADI layer receives a message with the handler pointing at this node

The first one will trigger an content shift from the queue manager to the application buffer with required contents in the queue, and the second one will register a description in the MPICH runtime manager, which enables that when the application calls for the content from receiving queue it can be shifted to the user buffer directly.

Another queue the ADI manages is the device ranking queue, and this queue is initialized by the user when the distributed application is deployed. It contains all the devices the MPICH runtime manager have access to, and abstract the invoker as an communication interface. The application do not need to know the existence of this list, and by invoking the communication interface, runtime manager will post the messages according to the pre-configured ranking list after an outgoing handler is created. When receiving the queue, runtime system will fairly threat all the resources, and no priority in receiving queues. The ADI uses an round-robin style of circulating query around the devices and asks if there is a new message(Protopopov and Skjellum, 2001).

RDMA channel implemented on ADI3

MPICH introduces ADI3, the Abstract Device Interface version three, which enables programmers to implement their own communication strategy among nodes. The structure of ADI is consisted of two parts, and can be presented in the following fashion:

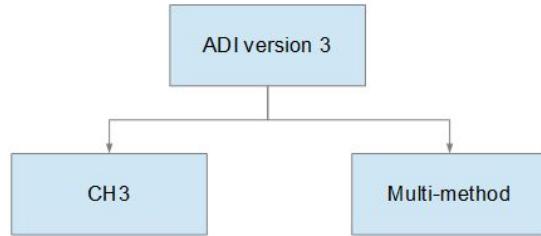


Figure 2.11: Structure of ADI3

The CH3 is a bundle of functions which implements the interfaces for different format of communications, including TCP socket and SHMEM channel. The reason to abstract these interfaces in a subclass of ADI3 is that the CH3 bundles them into *channels*(MPI-CH, n.d.). The channels are based on normal Unix based socket, and introduces features such as queuing and listeners to achieve better performances.

The implementation of RDMA over MPI-CH structure is introduced via this mechanism, because the ADI is consisted of a set of macros and functions which allows varieties of different software-hardware infrastructure to be implemented. The design of this mechanism allows the connected hardware to implement their own message queues and data processors, which is required by the RDMA verb mechanism which overrides the kernel model in UDP socket processing and memory management by introducing its own network data exchange protocols. The RDMA channel only contains five functions, and will be discussed in the next subsection.

RDMA channel implementation

RDMA channel is designed for global memory share mechanism. It contains two functions for communication, *put* and *get*. Other three functions are for process management. Shown in the discussion of RDMA above, it maintains a queue between two nodes separately. Both *get* and *put* are non-blocking, and the data they manipulate can go directly into the queue rather than wait until the entire process to finish. It is easy to see that the actual implementation of these two functions are different from RDMA standard description, whose communication methods are one-sided but the actual implementation in CH3 is two-sided.

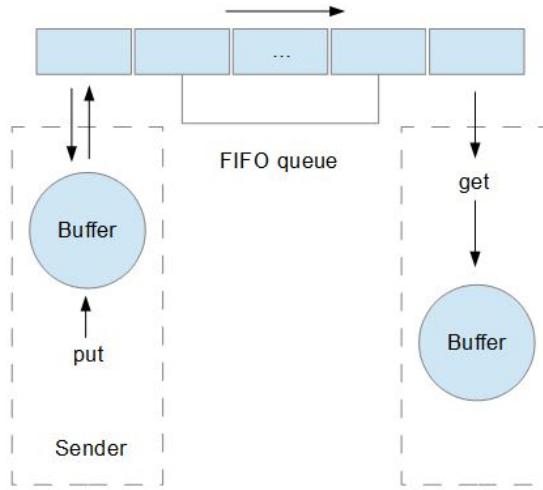


Figure 2.12: The message queue between nodes

The put-get pair can manipulate the buffer in a straightforward fashion, and the memory buffer can be viewed as an circle: the pair holds an fixed size buffer with an fixed head index, and the put method can insert contents into the circle start from the head buffer, meanwhile the get method invoked by the receiver can read contents also start from the head index and points at the tail. Therefore, the buffer.read method, which is time consuming, can be asynchronous with the buffer.write, as long as the buffer is not full. This way of communication also makes it easy to synchronize the memory addresses between the sender and receiver, because the communication buffer uses only one header index and both put and get starts from the same place, only with the opposite direction in the circle.

The reason of implementing an independent RDMA channel over CH3 rather than creating a new subset of communication protocols of ADI3 is:

- Modern implementations already have similar shared memory mechanism in CH3 layer, and the improvements in a subset of CH3 can inherit the interfaces of CH3 itself, and other system built purely on CH3 can benefit from it
- Potentially, other MPI mechanisms will be introduced into the MPICH implementation, for example the Infiniband RDMA which discussed above that it has one-side communication method, and the new implemented methods can use the RDMA channels by following the CH3 interfaces
- The portability requirement of MPICH design makes it necessary to consider a general design of communication channel rather than a specific one

The fat-tree topology

In this project, the RDMA channel performs as an optimized data exchange channel for the received data, however, the CH3 does not define the way RDMA channel organizes the network topology.

As discussed in the RDMA section, the best practise for RDMA performance is to implement fat-tree topology within the cluster, especially under the circumstance that nodes are not powerful enough to process complex computation tasks, therefore the design of software implementation over the cluster needs to sacrifice the bandwidth resources to achieve better overall performance with ARM based CPUs.

The fat-tree topology pairs the dual-feed receivers as leaves of the bottom layer, along with an upper layer node also supported by ARM CPU, the added bandwidth among the three nodes is equal to the output.

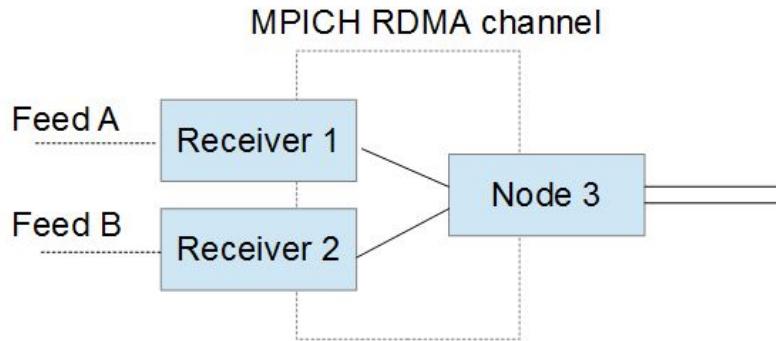


Figure 2.13: Fat tree implementation

We expect the RDMA in MPICH can reduce the requirements computational power of each receiver, and the parent node $node_3$ can have better hardware support to perform more complex algorithms compared to the receivers.

2.4.3 Conclusion

This section explains the logical structure of the system to implement the dual-feed receivers. Using the MPI-CH implementation under two parallel Raspberry Pi hosted applications, the system can achieve the goal of receiving dual feeds without implement expensive hardware which solves the problem by implementing parallel hardware bus between the CPU and two or more NIC cards. This section also explains how the three important parts works in the background: message queues, remote memory access implementation and fat-tree network topology. The next section will talk about how the third Raspberry Pi works as the master node connecting to a FPGA.

2.5 Field Programmable Field Array(FPGA) Assisted Computation

This section introduces the advanced hardware implementation on the $node_3$ discussed above. The Field Programmable Gate Array, FPGA is traditionally considered independent from the general computation architecture such as the CPUs, however, recent development in hardware pipeline technology enables the FPGA to communicate with the general architecture directly, and the content swap from FPGA to CPUs are much faster than the network supported data transmission.

FPGA is known for handling single threaded tasks which are hard to be split into sub-tasks, or the resources for parallel computing is much more costing than those for the single threading processes. FPGA performs all the computation tasks via hardware directly, and they are programmable.

The project implements FPGA to support the *merging* process after the cluster receiving the data feed. Discussion below shows the details of following technologies:

- Sparta-6 integration with Raspberry Pi
- FPGA programming technology
- Communication between the CPU and FPGA: wishbone bus

2.5.1 Sparta-6 integration with Raspberry Pi

We put a Sparta-6 FPGA chip over $node_3$ which is an Raspberry Pi2 board. The firm ValentFX has created an integrated environment for the FPGA to run on the Raspberry Pi boards on Linux based systems.

First we need to understand the differences among programs running on three different platforms: CPU platform, micro controller platform and the FPGA platform.

Programming on CPU platform

Programs running on the CPU platforms are the easiest to understand: the programmers, in most cases do not need to worry the details of electric signals inside the CPU chips, and they only need to focus on the high-level logic such as giving an integer value to a parameter a and add one to it for ten times. The programmers do not know how a looks like in the CPU and how the logic 'add one to a for ten times' works in sequence in the accumulators, data swapping in first level caches and system clock triggers.

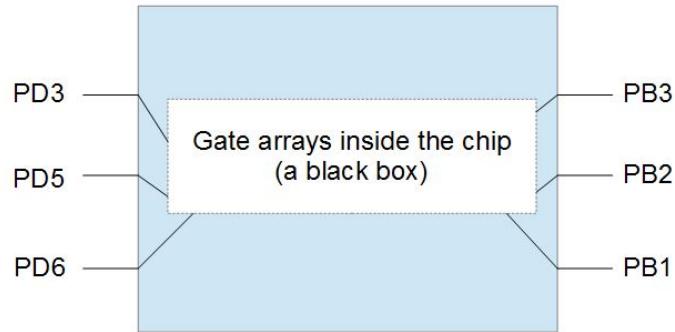


Figure 2.14: Arduino MCU

The MCU platform

The micro controller platforms are similar to the FPGA ones, and in modern times, the MCU producers will release tools called MicroController Abstract for the programmers to *translate* the high-level programming language, normally C, into the hardware languages that micro controllers can understand. Programmers know the architecture of the MCU their programs run on, and uses modern programming languages to manipulate the signals and clock triggers. Both the input and output of the MCU can be *translated* by the MicroController Abstract layer into the high level programming format. Take the Arduino MCU platform for example: the MCU can produce six modulated electric pulses on its six pins showed in picture 2.14.

An example code for setting and reading the a value from pin *PD3* is:

Listing 2.5: Reading PD3 from Arduino MCU

```
void setup() {
    Serial.begin(9600);
}
void loop() {
    int sensorValue = analogRead(PD3);
    Serial.println(sensorValue);
    delay(1);
}
```

The MCU works in this way: in the setup phase, the program set serial frequency to be 9600 bit per second as the data frequency of the pins, then the MCU is read for sending and receiving signals for each available pin, and in this example we read the default value of pin *PD3* by using the method `analogRead()`. This function in the hardware abstract layer will be translated to a pulse which goes into the circus, and gain the module of ping *PD3* signal within a system circle, and goes back to the hardware abstract layer. They layer will again translate the signals into a high level programming format, for example, if the black box attaches the *PD3* pin to a high voltage source of 5 Volt, the abstract layer will translate the module 5 Volt into an integer of one.

The expression of different signals can be fixed by the manufacturer or defined by the programmers, however, the key idea is that the black box which controls the behaviours of pins must be known to the system designers, and everything inside the black box is fixed and unchangeable, especially the modular logics within. Therefore, the design process of the MCU platform programs are:

1. Choose and understand the hardware logics of the MCU board
2. Design the software which uses the functions provided by the hardware

A good example are the MCU chips in the heat sensors, which receives the signals of sensors it attaches to and the outputs are distance from the source to sensor, and a pair of coordinates indicating the relative positions of source. Programmers can make use of the outputs to design their programs, varying from an intelligent flushing toilet controller to navigating program in heat guided missiles.

The FPGA platform

The FPGA and microcontroller platform is quite similar, as both need to know the detailed hardware implementations, and a hardware abstract layer to translate the programming languages and runtime signals.

The difference between the two is FPGA platforms do not have a *fixed* module logic showed in the picture 2.14, and the designing process of FPGA platform has two phases:

1. Design the module logics
2. Design the software which uses the module logics in the FPGA

This process looks complex in the first place, because the module logic in FPGA chip is designed by the programmer, so the software needs to contain a full implementation of hardware abstract layer to perform the bi-directional translation. Programming on FPGA was difficult in the early times, however, the modern design of FPGA boards absorbs the advantages in MCU and FPGA, and creates a structure illustrated in the following picture:

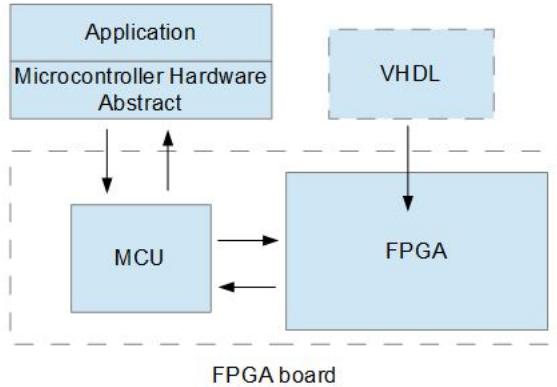


Figure 2.15: Modern FPGA design

The programmer nowadays do not need to worry about the hardware interfaces of FPGA chip, as the manufacturers have implemented an MCU over the FPGA, and setup the interfaces via MCU instead of the FPGA directly. As discussed above, the MCU has fixed interfaces, and the MCU on the FPGA board provides fixed general interfaces for applications to invoke, and fetch the data from FPGA. The advantage of using this middle ware structure is that the programmers do not need to care about the hardware implementation of the FPGA itself, and manufacturers can provide general drivers for the MCU over specific operating systems, with which the FPGA computation resources can be abstracted into a function in high level programming languages.

Spartan-6 with Raspberry Pi2

The Spartan-6 FPGA over Raspberry Pi, the Logi Pi follows similar way, and it does not use the MCU but rather a technology called to create a seamless communication between the CPU and FPGA. It uses Spartan-6 FPGA chip the hardware interfaces to connect to the Raspberry Pi board, supporting Linux based system, and programs can be burnt into the FPGA directly via general tools.

The advantage of using Wishbone will be discussed later, and for introduction, the Wishbone abstracts the MCU discussed above for running programs in FPGA and communicate with CPU. It is an abstract layer which now support Python and C programming language.

2.5.2 FPGA hardware programming

As discussed above, the FPGA programming involves two aspects, the software part which communicates with the FPGA, and the FPGA module logic which simulates the function of a MCU. This section will discuss the latter part, and show how this project works with

the FPGA hardware design.

Program the hardware logic module in FPGA forms the following steps:

- Understand the board structure: input and output signals
- Design the hardware logic inside the FPGA chip
- Programming the logic into FPGA and test

Board structure

The first step is to understand the board design, including understand the hardware resources we have on the board. For example, we need an accumulator to perform a for-loop, however, we need to know which pin refers to the clock on board, to trigger every step of the loop. In the Logi Pi v1.5 board, the clock can be triggered by an oscillator with standard 50MHz signal source, and connects to the *P85* pin.

Usually, the pins of the FPGA chip are bounded into banks, so the electric schema of FPGA chips will show the pin structure as a form of banks. The FPGA manufacturers do not know the components outside the chip itself, however, only the creator of FPGA board can know how the board structure looks like. The design of FPGA application functions needs a file describes it, to avoid checking schematics diagrams every time, including the number, name and function of each pin on the FPGA chip, therefore we can use the hardware description language(HDL) to implement our functions.

Design the hardware logic

The second step is to understand the function need to implement: how the signals are imported into the FPGA, and how the output is consumed. The signals are variables that are created in the applications running on the CPU platform, and we have the Wishbone mechanism to slurp the signals from the memory in CPU to the SDRAM on the FPGA board, and push them into the designed pin.

Design of the hardware logic in FPGA follows the idea of modular programming, and the designer needs to split the hardware design into basic modules according to the functions FPGA chip provided. Each module is consisted of three parts: the input, which can be a system signal like clock, or the output from another module; the basic logic which contains the hardware logic of how a signal is processed by the basic components, for example the pulse-width modulation logic to carry a constant signal with a periodic clock using the following logic:

1. Defining the input signals with fixed bits
2. Defining the parameter which receives the system clock signal, we can treat this parameter as an instance of class 'clock', which will change the value of itself according to time, and have methods to trigger an event which can be captured by system

3. Defining the output, including the bits it contains and pin number it attaches to, the mapping of pin number and actual function, for example, which pin attaches to the LED, can be found on the schematics diagram
4. Making a loop triggered by the clock, and define the behaviours of output inside the body

The example code shows how to carry an eight-bit long constant signal with the provided system clock, and can be presented in the picture 2.16. This diagram is the basic component of the hardware design, which we introduce to illustrate how the basic components discussed above are connected together as an overall hardware logic, to perform certain algorithm.

The benefit of using the modular design is that the modules are reusable, and the basic logic components inside FPGA is barely changed, so we can always reuse some classic solution even the code is decades ago.

The Logi Pi FPGA chip supports most of the modular design in the VHDL library, and the manufacturer developed some customized interfaces mainly for chip-to-chip communication especially for the ARM based platforms via PCI ports. These interfaces are the basic parts of this project, called Wishbone and will be discussed later.

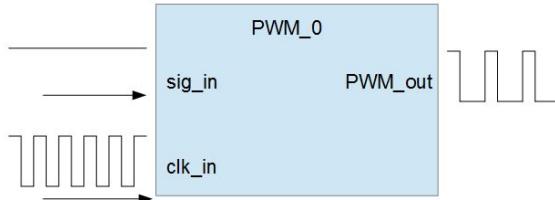


Figure 2.16: PWM modular logic

Burning the design into the FPGA chip

The FPGA chips are reprogrammable, and the programmer needs to make the chip aware the design using a hardware-based descriptive language to make the FPGA chips know how to form the proper hardware structures. There are several different hardware description languages, and the main stream are the Verilog and VHDL. This project will not introduce or compare these two HDLs, as both are well-known and widely implemented over both academic and commercial fields. Most of the Logi Pi modular interfaces are designed using VHDL, and the project needs to reuse some of them, we choose VHDL as our programming language.

The hardware description language abstracts the hardware details, to improve the universality of the designs. Therefore the module design of input/output signals need a mapping procedure to find the correct hardware resources they combine to. This mapping is based on the descriptive file of the board mentioned above.

Design of the hardware logics always include reusable modules, and it is less likely that designers need to focus on the implementation of a simple module like PWM modules, and a library of implementation details is required when burning the design into the actual FPGA chips.

We have already known the board structure including input/output signal environments and the FPGA chip functions, we need to combine these information together to make the FPGA work as we designed:

- Write and test the VHDL described design off board
- Introduce the hardware description file with UCF extension to link the modules to hardware interfaces
- Introduce the library to complete the detail design

The process now can be performed by the integrated development environment provided by Xilinx ISE tools. However, as the Spartan-6 is an old product and the ISE design suite is no longer supported by the Xilinx, there will be some difficulty in compiling the code, and will be explained in the system implementation chapter.

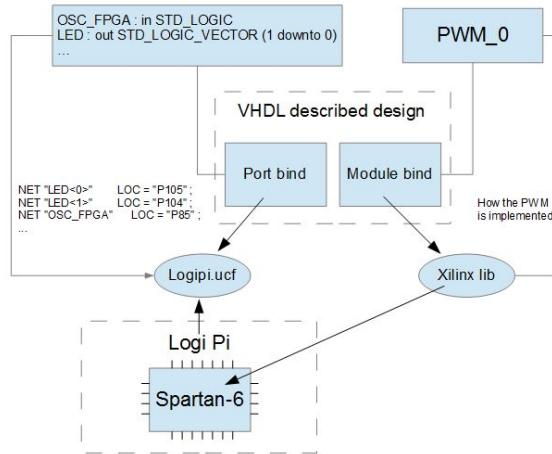


Figure 2.17: VHDL file structure

2.5.3 Wishbone: interface for FPGA-to-PC communication

We had a brief introduction in how the FPGA works in the above sections, however, the project requires communication between an CPU hosted application and a FPGA hosted one. An commonly used solution is the serial communication architecture, and the communication is via the universal asynchronous receiver/transmitter interfaces(Michael, 1992). The architecture is in picture 2.18.However, the UART technology is an overkill in this

project, as the Logi Pi FPGA board is connected to the Raspberry Pi via the GPIO ports, and we can use the serial peripheral interface bus to implement the communication instead.

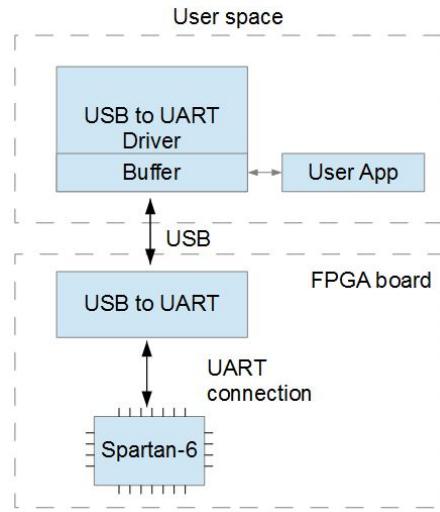


Figure 2.18: UART architecture

Serial Peripheral Interface on Raspberry Pi

SPI solves the problem of visibility of FPGA modular details on the application side. Without SPI, the applications need to know the exact hardware layers between the FPGA pins and the GPIO ports it connects to, and all the read/write performances need to call a chain of hardware data drivers to make the communication work. However, the SPI isolates the banks by tagging the FPGA pins with addresses, and ideally, the applications on CPU side only need to know the virtual address it connects to, and the SPI layer will deal with rest of the details.

The structure of SPI is quite simple, which is consisted of a master and a slave. The master is embedded on the Raspberry Pi board forming two pairs of communication channel, a slave select channel and a clock:

- Master Output Slave Input: data is the output of the master SPI and the slave receives the data
- Master Input Slave Output: master SPI receives, or reads the data from slave, however, the slave normally cannot invoke this channel
- Slave Select: the master SPI can select different slaves if there were multiple implementation of slaves in the FPGA
- Serial Clock: the master can output an clock signal instead of the build-in clock signal on the board

The Raspberry Pi disables the SPI driver by default(Org., n.d.*b*), however, it is easy to unlock it by using the tool *RASPI-CONFIG* provided by Alex Bradbury to unblock it(Bradbury, n.d.). The communication of SPI showed in schematic diagrams 2.19. The SPI slave port connects to both the Spartan 6 FPGA chip and the SDRAM chip on the Logi Pi board, and we can use the SDRAM to store big data.

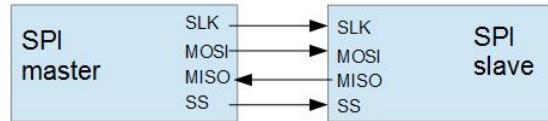


Figure 2.19: SPI schematic diagram

Implementation of Wishbone on Logi Pi

The SPI solve the problem of data driver using GPIO ports communication, however, the implementation over pure SPI is still too complicate to achieve. The application on the CPU side still needs to know the implementation details of the FPGA side, for example, the choice of SPI slaves is triggered by the application, however, different FPGA chip may have varieties of slave memory access address according to the compiling environment for the VHDL. Wishbone technology provides an alternative solution that wraps up the SPI interconnection details into high-level programming language libraries, and combined with address mapping tool provided, the applications on CPU side can have a relatively *fixed* call addresses.

The ValentFX has implemented Wishbone libraries in C and Python(ValentFx, n.d.*c*) with the above design, sacrificing some flexibility and providing much convince as designers do not need to reimplement any communication interfaces any more. The implementation of the libraries abstracts all the low level drivers and kernel details between the Raspberry Pi hardware and Logi Pi hardware, showed in picture 2.20).

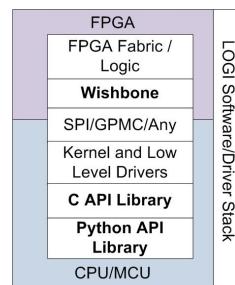


Figure 2.20: Wishbone register
(ValentFx, n.d.*b*)

The basic component of Wishbone is the register, which has a relatively *fixed* address usually starts with $0x0$ with an offset $0x1$. It is the basic interface for all the wishbone logics and has 16-bit input/output(ValetFx, n.d.a).

As mentioned above, the input and output processes are abstracted by the Wishbone wrapper. Take Python for example, reading and writing with register $0x0$ can be performed using the two functions provided by the library *logiRead* and *logiWrite*:

Table 2.3: Wishbone functions

Parameter	Function	Notes
address	logiWrite logiRead	the read/write destination address in hex format
high-byte low-byte	logiWrite	the write content in hex format and the string will be converted to ASCII. It has a pair of parameters because the registers are 2 bytes long
length	logiRead	define the byte it reads, 2 means read the entire 16 bits

This project does not have any peripherals and the hardware algorithms are implemented via the basic input/output processes. So the application on the MPI side needs to communicate with the registers via Wishbone, and the algorithms are performed after the data is sent to the registers. After receiving the data from Wishbone bus, we can implement the hardware algorithms afterwards.

The naming rules for the IO process in Wishbone follows:

- Each pin is abstracted into an address in the SDRAM, and the FPGA performs IO sessions with SDRAM instead of GPIOs directly
- The SDRAM session with SPI is two sided, and the SPI can perform both read and write to a pin by calling the same address
- The address-mapping is hardware irrelevant, therefore different SPI or Operating System implementation will not affect the CPU side applications

Parallel process and FPGA status

The FPGA processes are running parallel with the applications on CPU side, and the application design should take this into consideration. The processing speed of FPGA is much faster than the normal CPU hosted applications, and the clock frequency is different between the cluster. The Wishbone mechanism synchronise the clock via transmission layer, however, the application inside the FPGA still needs to wait for the read signal from the application.

Another problem is that the status of FPGA is controlled by the electric signals only, and the interfaces are fixed. Therefore if the application on the CPU does not read from the

FPGA first and change the status of FPGA, the processing result in FPGA modular will soon lost before the CPU applications can react.

These two features requires the design of the system to be aware of the execution sequences especially in the FPGA communication.

2.5.4 Conclusion

This section introduces how the FPGA will work with the receiver applications on the MPI-CH structure, and focuses on how to design and implement a hardware algorithm in the FPGA, from design to actual coding, and the important technology which enables the hardware to seamlessly communicate to the CPU hosted applications using Wishbone. The FPGA accelerated computation is widely used in the commercial solution to similar problems, especially in the market data feed handlers, however, this project shows the potential of using an open-source FPGA board to achieve the same goal over an cluster hardware environment.

2.6 Conclusions for the Chapter

In this chapter, we introduced the core technologies we implemented in the project, and focuses on the Message Passing Interface; the Remote Direct Memory Access method and the reason of implementing Fat-tree network with explanation of its benefits; the Hardware design and programming technology of FPGA and focuses on how the hardware can communicate with the software running in the modern CPU supported system.

We draw a blueprint of a hardware platform which is linked by Ethernet, and the software runs on it is supported by MPI. The system is scalable and efficient because we implement the Fat-tree network with adjustment of interconnection topology and bandwidth.

The following chapter will discuss the system design and the real implementation of the experiment, and how the features are brought to life.

Chapter 3

System Design and Implementation

This part of the thesis explains the main objective of the experiments, design of the experimenting system, and the implementations of technologies discussed above. This part is consisted of three parts:

Table 3.1: Content structure of system design and implementation

Name	Content
Experiment Objectives and System Requirements	Showing system requirements based on the hardware achieved, and simulation goals
Hardware Design and Implementations	Showing the hardware design of the system, including connectivity implementations and power supply implementation
Software Design and Implementations	Showing the design of different sender-receiver pair, with and without MPI layer and FPGA assisted ones
Hardware Algorithm Design and Implementations	Showing the design of the hardware implementation along with the software interfaces of the application on the Spartan-6 FPGA

3.1 Experiment Objectives

3.1.1 Overall Design

The project designed an experimental system over Raspberry Pi 2 Model B hardware, that supports 100Mbps Ethernet dual UDP datagram feeds as data source. The system should be flexible in error tolerance such as package loss, and scalable in implementing additional heterogeneous hardware. The network can be separated into two parts: the communication layer that supports low-latency network, and monitor system embedded in the SoC that implements additional features such as error tolerance.

The System over Chips(SoC) system are interconnected as a homogeneous processing cluster. The task/thread management are performed by interconnected CPUs over multi-processor interfaces, memory solution for bandwidth requirement of high frequency data is RDMA protocols implemented over open sourced hardware memories. The FPGA attached in the master node is in charge of decoding, decompression and filtering for dual feeds.

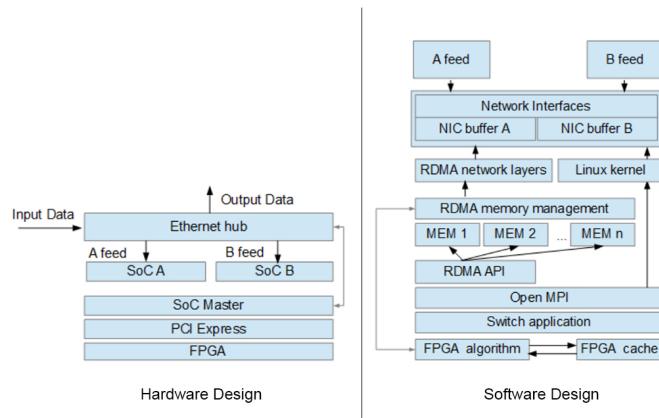


Figure 3.1: System blueprint

This solution implements and improves the solution provided by the commercial firms in 10Gb Ethernet Switches, which introduces high performance computation chips to resolve the pressure in the network interface buffer. The improvement is in the layer of the high performance computing: in this solution, the HPC layer is not linked directly to the NIC in the board, but is interconnected by the Ethernet Switch, which is much cheaper than 10Gb Ethernet switches. It sacrifices bandwidth in inner system to achieve similar efficiency in data processing by give spaces to heterogeneous scalable computational platforms, consisted by open sourced hardware like Raspberry Pi, and the fat-tree topology interconnection enables the cluster to expand without much cost.

The system can be implemented as the interconnection switch network for cloud computing clusters and high performance computer clusters. The design introduces the infrastructural solution for low-latency market data feed, utilizing data transmission efficiency with limited bandwidth, and the fat-tree topology network provides features in system monitoring and scalability in supporting massive computation.

This thesis provides a possible solution to this problem, which is implementing the dual data feed system over the open sourced hardware, and using Message Passing Interface technology to make the system scalable to achieve similar performance as the commercial ones with lower cost in purchasing and energy cost.

3.1.2 Main Objective

The main object for the project is to implement an experiment over a combined hardware-software environment with the features showing in table 3.2. The system is consisted of two parts:

- The hardware part: creating physical connection among ARM based SoCs and FPGA accelerated data processor, and connect to the data feed via Ethernet cables;
- The software part: creating logical fat-tree topology network, that buffering dual feeds in the slave nodes and processing UDP packages via FPGA accelerated master node and other parallel user cores;

Table 3.2: System Features

Features	Description
A-B UDP data feed handler	The system needs to support dual data feed(A-B data feed) to simulate the senario of ultra-low latency market data transmission
FPGA accelerated UDP package processsing	The decompression and filtering of dual UDP packages in FPGA are proved to be more efficient than pure CPU solutions, and can be expand to more general situations in normal high frequency data networks
ARM based SoC interconnection	Interconnection among ARM based SoC network makes it possible to implement scalable buffer capacity and bandwidth
Fat tree interconnect network	Fat tree topology is proved to be efficient in implementing interconnection network for massive parallel computation, and flexible in scalable computational nodes.

3.2 System Requirements

This project wants to achieve the overall following goals:

- Inputs of the system, including the dual-feed hardware structure, and UDP session simulation
- FPGA accelerated package filtering: the logic and hardware implementation
- Monitoring and printing the send-receive results of the communication

The project does not implement the hardware equipments of 10Gb Ethernet because of the complexity in constructing the hardware over Raspberry Pi and the server side. In order to simulate the environment using RJ-45 cables, we implement a simulated software environment which manually configure the package loss, sending/receiving window and other scenarios.

3.2.1 System Input: Dual feeds and UDP Session Simulation

This session explains the requirements in system input, including how to form the hardware input of dual feeds from the simulated server side, and the UDP session simulation, especially how to choose the package loss ratio.

Dual-feed receivers hardware

The main goal of this section is to assembling a hardware using Raspberry Pi boards that can receive two network signals from two RJ-45 cables. The bandwidth of both feeds are up to 500 Mb, however, this includes the bandwidth of incoming data and interconnection communication.

In order to measure the experiment results in an easy way, we simulate the server side with only one source, and the offset of package is manually set up in the sender application. Signal of the sender is split by an Ethernet Switch.

Sender-receiver pair requirements

The sender application should be able to differ the receiver with manual offset, and the receiver needs to be able to implement the following features respectively:

- Receive the data feeds and perform error-check without communication between the two feeds

- Receive the data feeds and perform error-check after merging the feeds, this requires communication between the nodes, and this should be separated into two scenarios: with and without MPI communication channel support
- Receive the data feeds and introduces the FPGA algorithm to perform the error check

UDP simulator

This section simulates scenarios of UDP transmission in low-latency network. According to the Cisco user manual of the commercial high-frequency market data handler, the metrics of the bandwidth and message rate follows(Jain, n.d.):

Table 3.3: Low-latency market data handler metrics

Name	RTT	Average Time Cost Per Hop [usec]
No watchlist overlap	1	6
Custome watchlist overlap	1	7

It is clear that the 10Gb Ethernet bandwidth and data speed is higher than the RJ-45 maximum metrics. The experiment, however, introduces the possibility in packet loss of UDP datagrams, and scaling down the size of required bandwidth and message rate with corresponding package loss ratio, to prove that if we implement similar solution over the 10Gb Ethernet using optic fibre with the same Raspberry Pi structure, we can still achieve the results close to the experiments.

The calculation of packet loss ratio of UDP datagrams P_{UDP} in single feed follows with the listing definitions(Moon, Kurose, Skelly and Towsley, 1998):

- N_{send} is the packages sent from the sender side
- N_{rec} is the package received in the receiver side
- Msg_i defines the status of a package i :

$$Msg_i = \begin{cases} 1 & \text{packet } i \text{ is received} \\ 0 & \text{packet } i \text{ is lost} \end{cases} \quad (3.1)$$

- $delay_i$ is the delay of packet i

Therefore, it is easy to understand that the average package loss ratio \bar{l} is $\frac{\sum_1^N Msg_i}{N}$, however, in this project the scenario is a bit different, because in high-frequency network, the receiver

implements a time-out more restricted to normal network threshold recommended(Eggert and Gont, 2009). The real package loss ratio in this project should be:

$$\bar{l} = \frac{\sum_0^N Msg_i - \sum_0^N H(delay_i - t_{threshold})}{N} \quad (3.2)$$

$$H(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases} \quad (3.3)$$

So the accuracy of the experiment result now lies in the estimation of normal package lost rate and package overtime rate corresponding to the chosen bandwidth, and simulate the environment of 10Gb Ethernet under the simulated experiment environment. Therefore, we can estimate the real UDP package reliability in low-latency network with the simulated package lost Msg_i and overtime packages whose $delay_i$ is larger than the fixed threshold in the receiver.

Value of the packet loss rate

According to a research in the UDP multi-cast network, the package loss rate in multi-cast network of UDP protocol can be expressed in the following way(Cáceres, Duffield, Horowitz, Towlsey and Bu, 1999): Given the definitions listed below:

- V is the overall network using the UDP multicast
- α is the given node set status when the node lost packet, and $\alpha = \alpha_k, k \subseteq V$
- χi is the i th independent observation result of a node, where $\chi > 0$ means it receives the package, and $\chi = 0$ means it does not receive it
- Ω is the subset of V whose nodes loses packages

We have the maximum likelihood estimation of packet loss in this network:

$$p(\chi_0 \dots \chi_N | \alpha) = \prod_0^N p(\chi_i | \alpha) \quad (3.4)$$

It will be convenient to calculate the overall package loss if we manually set the observation result of single node χi to a step equation, therefore $\chi i = \begin{cases} 1 & \chi = 0 \\ 0 & \chi > 0 \end{cases}$. In order to make the calculation easier, we take the logarithm of likelihood forming a log-likelihood estimation:

$$\eta(p) = \log \prod_0^N p(\chi_i | \alpha) = \sum_0^\Omega \log(p(\chi_i | \alpha)) \quad (3.5)$$

Therefore, we need to find out the measurement method of independent observation of package loss in each node $p(\chi_i|\alpha)$, however, this value only relates to two factors: the datagram size and predicted send time. These two factors are known to us from the above discussion measured from Cisco, and we need to find out an empirical formula to derive the value of it.

Considering a typical UDP send-receive process, we divide the possible scenarios into two:

- No package loss
- Package loss occurs

Given an equation of required bandwidth $B_{req} = \frac{S_{pack}}{T_{trans}}$, where S_{pack} is the total size of the packages and T_{trans} is the estimated time transmitting the whole package. We introduce another time variable specifically for the propagation time T_{prop} . In the above two scenarios, we have:

- No data loss: $T_{total} = T_{prop} + \frac{S_{total}}{B_{send}}$
- With data loss: $T'_{total} = T_{total} + T_{resend}$, and we have T_{resend} equals to the time cost in resending UDP datagram

$$T_{resend} = \frac{S_{total}}{S_{datagram}} \times p(\chi|\alpha) \times T_{prop} + \frac{p(\chi|\alpha) \times S_{total}}{B_{send}} \quad (3.6)$$

Assuming that UDP datagram lost will happen in most cases, therefore the second scenario can be viewed as the average value of T_{total} . From table 3.3 we have:

- Minimum requirement: $S_{send} = 45B$, $T'_{total} = 6\mu sec$, $T_{prop} = 1\mu sec$
- Maximum requirement: $S_{(send)} = 45B$, $T'_{total} = 7\mu sec$, $T_{prop} = 1\mu sec$

The time consumption for resending the package is relatively small, and $\frac{p(\chi|\alpha) \times S_{total}}{B_{send}}$ can be omitted. We assume that the processing speed is similar in both commercial environment in 10Gb Ethernet and the experiment environment, and the fixed number is t_{trans} . The data handlers use 2 Intel Xeon (X5570) 4-core 2.93GHz processors (Jain, n.d.), and the estimated processing time is $t_{trans} \approx 5.8\mu sec$, and $t'_{trans} \approx 6.82\mu sec$ therefore we have estimation:

$$p(\chi|\alpha) \subseteq [2.639\%, 3.448\%] \quad (3.7)$$

Choice of two receiving strategy

There are two different receiving strategies according to the distribution of filtering and error checking of UDP packages. Filtering is the process for checking each piece of data feed datagram i :

- Has the receiver received datagram $i - 1$

- Is the content in datagram i expected

and error checking is the process of merging the outputs of each data feed, and decide if we have received the expected content.

The first strategy is to implement filtering in each data feed receiver, and error check happens in the controller:

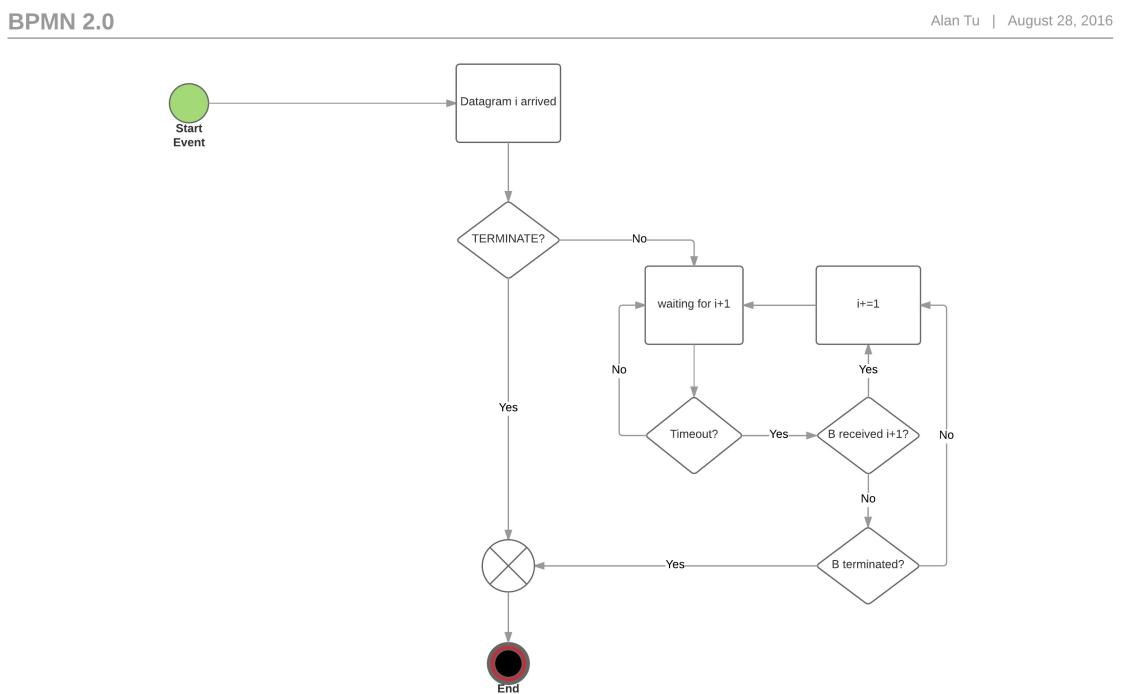


Figure 3.2: Flow chart for filtering in the data feed receiver in BPMN

For each data receiver A and B, we need to wait each to finish, or time out, then the results will be retransmitted to the controller. The filtering takes time, and the status query between the receivers will affect the efficiency of the system. Another potential harm is that one or more data feeds lost connectivity and the alive session needs to wait until the dead ones expired then to retransmit. This will significantly increase the latency in the controller, as the open sourced hardware is not reliable as the commercial ones, and the system needs to be able to tolerate this situation.

We implement the second choice: this one is to perform filtering and error check both in the

controller, and the receiver only in charge of individual feed, and the data will be retransmitted to the controller right after the session is over. This solution solves the problem in status queries among receivers, as they do not need to know the status of others, and dead connections will not affect the overall efficiency, as the controller hosts the keep-alive sessions among nodes parallel to the receiving sessions. The worse expired receiver is the one dies in the last package, and the controller only needs to wait for $T_{alive} + T_{offset}$, where T_{alive} is the maximum keep-alive time for each node, and T_{offset} is the manual offset between different data sources.

Conclusion

We derive the data loss ratio in high-frequency network for UDP datagrams should be [2.639%, 3.448%], with the following assumption:

- The data loss rate in UDP datagram is similar in both experiment environment and real scenarios
- The ratio of data processing time and propagation time can be considered fixed if the size of datagrams are similar

We also makes decision of how the data feed receiver works, especially how the filtering and error check process is distributed. We decided to distribute the filtering and error check in the controller together, and the receiver only needs to retransmit the received datagrams regardless the status of other receivers.

3.3 Hardware design

The table of materials used in the project is shown in table 3.4. However, we do not have

Table 3.4: Materials used in the experiment

Name	Amount	Manufacture	Metrics
Raspberry Pi2	3	Raspberry Org.	1x Boardcom ARM CPU, 1x Ethernet Port, 1x SD card port 5.1V 1A input power
Kingston MicroSD card	3	Kingston	8GB Micro SD card
Logi Pi FPGA board	1	ValentFX	1x SPARTAN-6 FPGA chip, 5V 1A input
Ethernet Switch	1		1x S1786 central chip, 8x Ethernet ports
Ethernet cable	4		RJ45 copper cable
Power source	1	iClever	6x Powerports, 100-240V 50-60Hz 1200mA input, 5V(max) each output
Power cable	4		USB power cable

3.3.1 Interconnection Design and Implementation

The Raspberry Pi only has one Ethernet Port, and data traffic among them can be achieved using two ways: the Ethernet cable and GPIO ports. The Ethernet cable way is easy to understand: Raspberry Pi boards forms a LAN and they can reach each other using Internet protocols. The GPIO ports is another ideal way for our project, because the data transmission ratio will not be limited by the interconnection architecture which overlays the incoming data feeds. This uses GPIO hardware communication structure which has been introduced in former chapter, and the applications in this solution exchange direct bit data via the hardware directly through the GPIO pins.

However, the implementation of UART interconnection has the following problem: the pin number of GPIO ports on Raspberry Pi boards is not big enough to process large quantity of data, and an solution for that is using an GPIO expansion board. We also need to implement an synchronizer between two GPIO ports because the system clock on different Raspberry Pi boards is slightly different, and will affect the efficiency in transmission.

Interconnection design

In order to perform the experiment with minimum cost, we decide to use the Ethernet connection solution among the Raspberry Pi boards shows in picture 3.3.

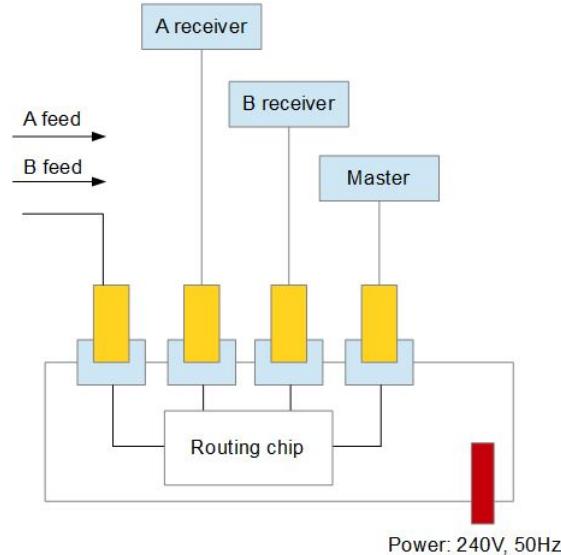


Figure 3.3: Interconnection hardware blueprint

The bandwidth requirement has been discussed in the previous sections, and we can derive the overall bandwidth of each system:

- Each port supports maximum 500Mbps
- RJ-45 cable only supports 100Mbps bandwidth(Kunz and McCamey, 1999)
- The bandwidth inside switch can be considered as infinite, as it is inner-chip communication inside.

Therefore, the actual bandwidth limit in one-side communication is 100Mbps, however, the MPI-CH is a two-side communication among the nodes, and the server input also uses the same interconnection network. Therefore, the upper limit in actual one-side communication is less than 40Mbps.

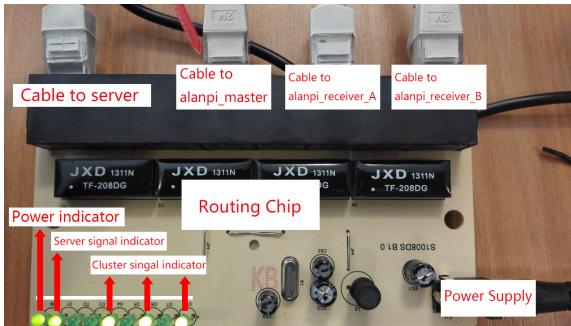


Figure 3.4: Cabling on the switch board

Interconnection implementation

We have two sorts of Raspberry Pi boards, the slave nodes that perform as the receivers, and a master node which connects to an Logi Pi FPGA board. The network connection of these two kinds are similar:

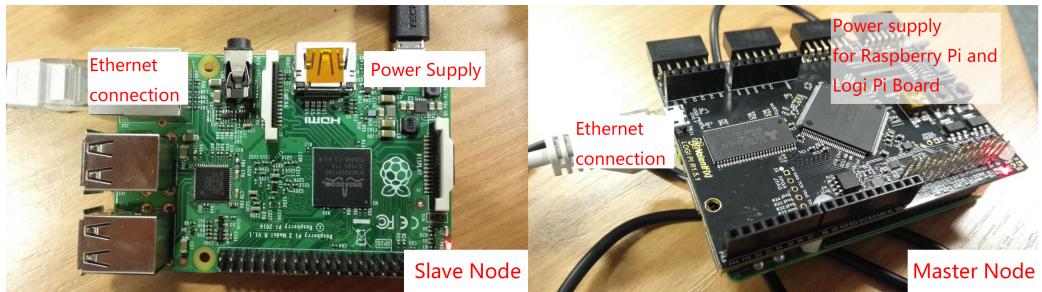


Figure 3.5: Ethernet connection in slave node and master node

3.3.2 Power Supply Design

The power supply of the cluster is divided into two parts(the simulated server has independent power supply). The switch is supported directly from a power strip without backup, and the cluster is supported by an USB power hub with stabilizer. The design should be enough for experiment, however, as the Logi Pi FPGA board does not have any static memory module, the hardware content will be lost if the power supply is cut off for the master node, we should implement Unstoppable Power Supply units for large scale implementation.

The design of power supply is illustrated in the picture 3.6. The FPGA can get power from both the GPIO pins in the master board, and the USB power hub with a standard A-B USB cable. The FPGA board is used for accelerating the processing speed and there is no meaning to keep it alive if the master node is off-line.

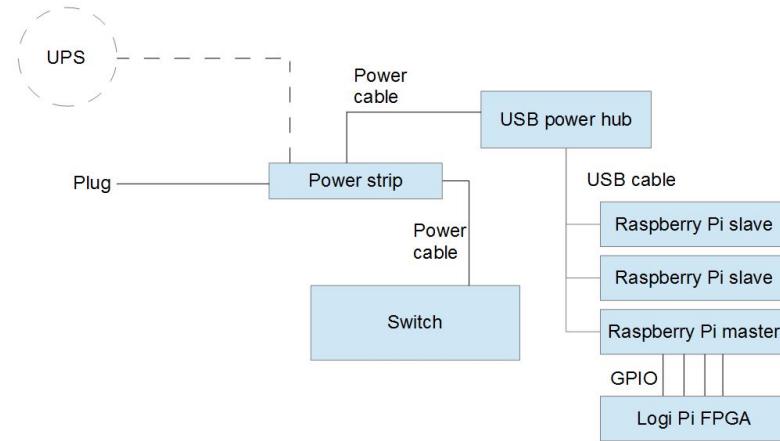


Figure 3.6: Power supply blueprint for the cluster

3.3.3 Conclusion

We discussed the hardware design in this section, and set the bandwidth of the experiment environment to be $B_{exp} \approx 40Mbps$, and the detail description for the network structure implementation and power supply implementation prove that the experiment over Raspberry Pi-Logi Pi cluster can demonstrate the solution we provided for the high performance network receiving dual feeds.

3.4 System Design and Implementation

The blue print of the software system over the cluster is illustrated in picture 3.7, and the implementations are focused on the MPI supported Linux cluster over Raspberry Pi, including the installation and configuration of MPI-CH2 framework on Linux, configurations of Python support and FPGA development configuration.

The Raspberry Pi hardware, however, is different from general computing platforms in the following ways:

- Only limited operating system images can be installed on Raspberry Pi, and most of them are deep customized for the ARM based platforms
- Hardware portals on the Raspberry Pi is different, including the SPI and i2c configurations

We encountered some legacy problems from both the cluster implementations and Windows system as the server. The solution for these problems will be discussed in this section.

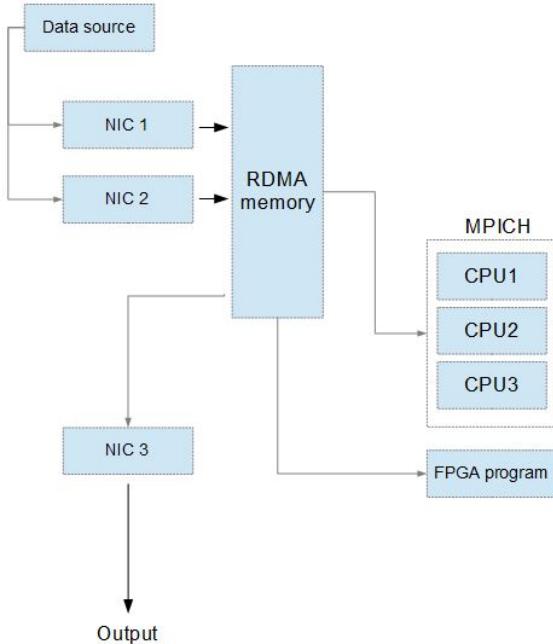


Figure 3.7: System blueprint

3.4.1 Operating System and Internet Connection

We chose the Raspbian from one of the official supported customized OS images by Raspberry Pi organization as our operating system. The Raspbian is customized from Debian to support the ARM CPUs, and other unique hardware drivers in Pi boards(Harrington, 2015). Writing the system image into SD card and plug it into the Pi board, the system is well supported by the community that starting up is quite easy. However, it would occur some bugs in the system if we do not start up the cluster in the correct sequence.

Solution to the IP address configuration

The Internet connection of the cluster is provided by the computer served as the sender. It shares the WIFI public network with the cluster using a wired cable linked from the computer to switch. After sharing the network, the IP of wired network adaptor in the computer will be fixed to 192.168.137.1 with net mask 255.255.255.0. We configure the Raspbians to manually get their fixed IP address on eth0 interface by changing the booting configurations, and the three nodes uses 192.168.137.101, 192.168.137.102, 192.168.137.103 in order to avoid conflict. This setting needs to be set in both booting configure and the image level file cmdline.txt.

Solution to Internet reconnection

We need Internet connection over the cluster to perform package installation and update, however, a bug in Windows 10 wireless network sharing policy can cause a lost in Internet support of the cluster. The bug happens when the OS is turned to sleep or when wireless signals are cut off for a while. The wired adapter will be turned into the primary adapter on the above scenarios, and some parts of the network sharing will not work even we manually reconnect the wireless network again.

A solution to this is to perform ARP scan from the server over the clusters manually after reconnecting, and be sure that all the interfaces on the Raspberry Pi are set to manual configuration, because the wired adaptor may lose the setting of disabling DHCP, and some network interfaces might be activated by mistake.

Interacting with the cluster: ssh terminal and VNC

We choose two ways to interact with the cluster, ssh using Putty client and VNC to see the XWindows desktop. The latter one is implemented by the RealVNC viewer on the monitor side(corp., n.d.), and tightvncserver on the cluster side(org., n.d.c).

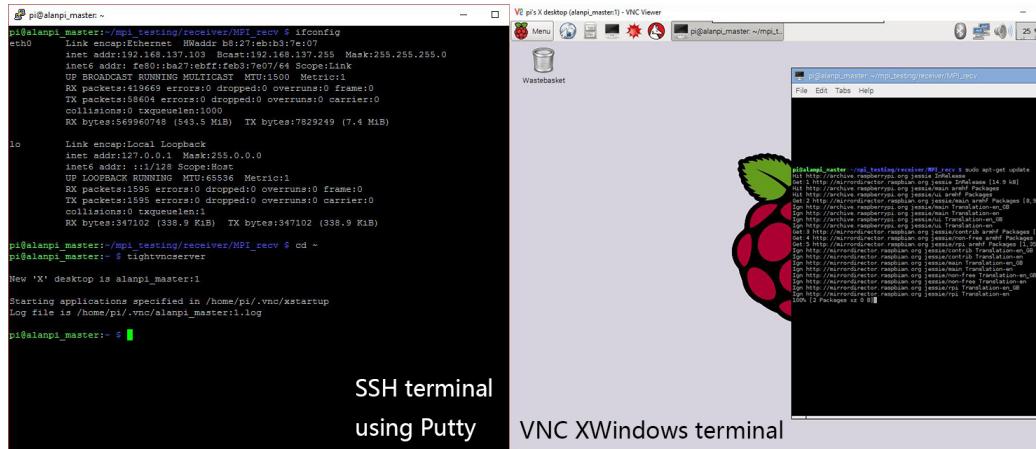


Figure 3.8: SSH terminal and VNC XWindows for master node

3.4.2 MPI-CH2: Install and Configure

MPI-CH2 needs to be compiled and installed from the source. The latest stable version for MPI-CH2 is 3.2. Installing of MPI-CH2 is time-consuming, and takes about three hours to compile, build and install the entire system without any error. Therefore we made one image with MPI-CH2 installed, and wrote that to all rest two nodes to save time.

This section will not discuss in detail about the installation steps and configuring system files, however, we can describe the overall process is to download source, build and compile, then add necessary middle wares, in the end the MPICH2 package provides example codes and will work if the installation is completed. The configuration of MPI-CH2 over Raspbian is a bit different from other x86 Linux images. The hostname of Raspbians are the same due to the default configuration, and changing the host name in the file /etc/hostname will affect the environment because some programs, including mpiexec and mpirun, will use etc/hosts as the indexing. The changing in IP address discussed above will make these applications unable to find the local host, as it is set to be 127.0.0.1 with the name 'localhost'.



Figure 3.9: Setting up the MPI-CH2 environment

Adding Python support to MPI-CH2:MPI4PY

Python has implemented MPI support in the language, in both Python 2 and Python 3(*org, n.d.d*). However, the library temporary supports only the OpenMPI, and we needs to manually add it to the MPI-CH architecture. Fortunately the MPI-CH has already prepared this scenario, and the MPI command environment supports the method `mpirun.openmpi` to execute applications using OpenMPI.

The Python MPI library, MPI4PY

3.5 Software Design and Implementation

The data flow blueprint for the project can be illustrated in the picture (?), and the implementation details can be very different according to the framework we are using. The software system can be split into three parts:

- Data feed: simulates the high-frequency data, can be configured into different sending modes with related package size and error rate
- Dual-feed receiver: the receivers are running over the MPI-CH framework, however, it can have two different strategy in processing received data: the error check mechanism happens in the master node, and the receivers can choose either to retransmit every single datagram right after receiving it, or retransmit the entire package after the transmission is time-out

- Master node: the master node needs to perform at least two functions: metrics collection and error check

We demonstrate the solutions to these requirements in the following contents. Multiple implementations of data feed and master node are introduced in order to make comparison of combination in the experiment, and we make decision of the strategy in the dual-feed receiver for some experiments done on the cluster.

3.5.1 Design of the Data Feed

The flow diagram of data feed is picture 3.10. It does not fully simulate the real A-B data feed over UDP protocols, however, because we send serials of integer data to the receivers in order to perform benchmark of the efficiency.

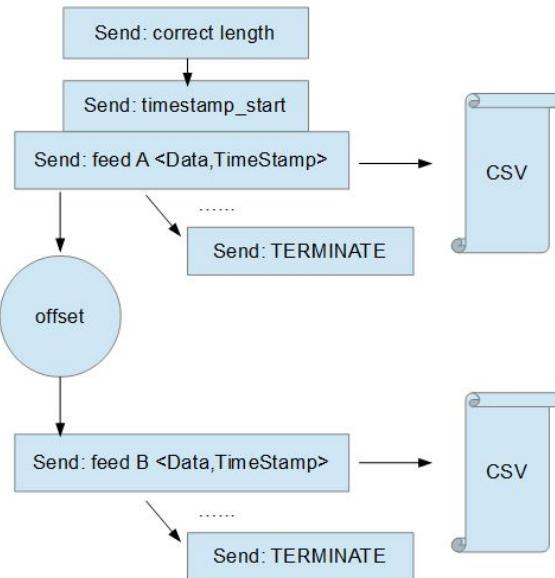


Figure 3.10: Data feed application blueprint

The main goal of the data source provider is to simulate the data loss, and we have two ways to implement it. One is to make a real manual data loss in some of the packages before sending, and leave the error check to differ the 'correct' data source and 'corrupted' data source. Another is to pre-set a bulk of independent sending tasks, and manually disable some of the tasks randomly.

The first option simulates the UDP datagram transmission directly, however, it takes time to make comparison, as for each sending task, the receiver side needs to run a full error check. It also cannot perfectly simulate the situation with one of the source significantly delayed, because the UDP datagram can arrive in mixed sequence and the UDP receiver will automatically sort the data according to the header.

The second option uses failure rate of the independent replications to substitute the data loss rate in single transmission, with the definition of Bernoulli experiment(Anderberg, 2014). We choose this design as the data sender.

Transforming data loss rate to session failure rate

For a group of tasks with an amount of N , the failure rate $P = \frac{N_{fail}}{N}$, and for a single task, the failure rate $p_{fail} = P$ therefore, for N times of experiments, the probability of failure happens k times is: $C_N^k(1-p)^{n-k}$, $k \in [0, 1, 2..., N]$. According to the discussion in system requirements, we have the data loss ratio $p(\chi|\alpha) \subseteq [2.639\%, 3.448\%]$, and we transform this ratio into the success rate of N times Bernoulli experiments by taking the $p(\chi|\alpha)$ into the equation of independent experiment probability.

The reason for this transforming is that the UDP protocol does not have an error check mechanism defined, and it is hard to benchmark efficiency in different data loss rate by comparing the success rate of transmission, as one data loss is a failure defined in the protocol, and the output of experiment i for one time A to B UDP session with data loss is either success, or failure. Instead of measuring the success probability, we made a bundled independent experiments, and randomly fail some of them to simulate the data loss, and the receiver do not need to implement the error check mechanism as the lost data will be filtered before receiving, which saves a lot of time.

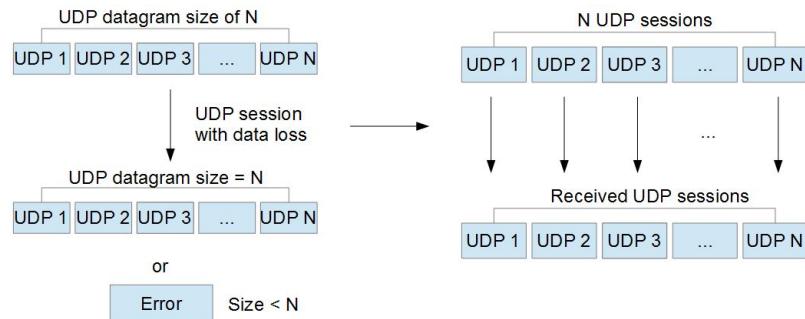


Figure 3.11: UDP sender from data loss rate to success rate

Dual data sender implementation

The dual data sender needs to implement the following features:

- Initiating the experiment, including sending beginning signals to the receivers and receiving environment variables
- Triggering dual data feed via UDP session to the receivers in two Raspberry Pi SoCs

- Triggering data collection process in benchmark applications

The first feature requires us to be able to send UDP packages to the MPI hosted applications on the cluster before the actual sending process happens, and the initialization should be able to take parameters associate to the experiment we performed.

The second feature requires us to be able to start UDP sessions to two different UDP listeners, and send packages simulating the A-B data feed discussed before, with fixed offset in latency.

The third feature requires us to be able to achieve two things: recording the time stamp in each individual session, and sending the recorded results to the benchmark using TCP sessions, as the data would be too big for the UDP protocol to handle.

We uses Business Process Model and Notation to show the flow control of our application(Dijkman, Hofstetter and Koehler, 2011):

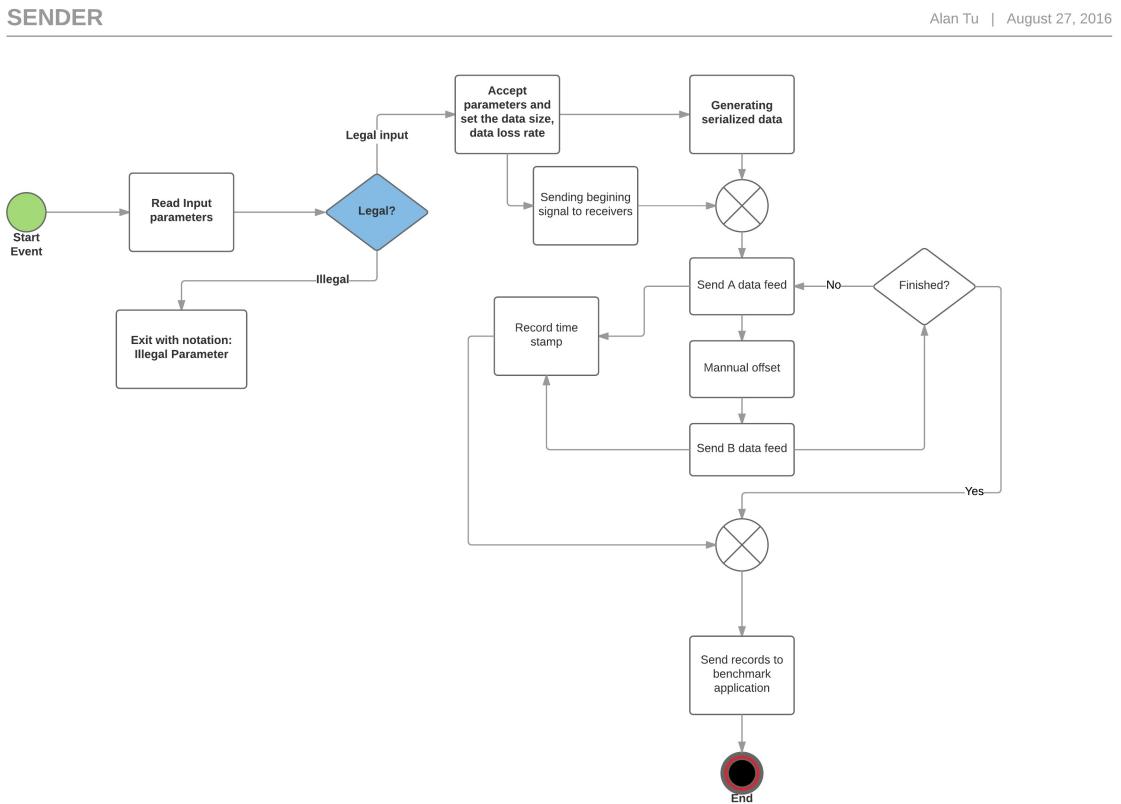


Figure 3.12: Dual sender flow chart

3.5.2 Dual-feed handlers and controller

The dual-feed receiver and controller is running on the Raspberry Pi cluster, consisted of three parts and on the MPI-CH2 framework. It has two feed handers, receiver_A and receiver_B, and a controller running on the master node.

We had two options to design the handlers and controllers over the MPI-CH2, one os using SPMD which only implements one application defining three behaviours, and the *mpirun* take only one applications running on multiple nodes. Behaviours and communications are defined by node rank and tag respectively.

Another solution is using MPMD which defines two different applications, one is for the handlers and anther for the controller. Therefore the *mpirun* needs to manage which node implements which application by defining the details in the *machinefile* required by the method.

We choose the first solution for three reasons: a united application is easier for coding and testing, and easier in scalability because we can changing the functions of each nodes by reordering the ranks defined in the configuration file of MPI-CH, and the version control over the cluster is much easier than the MPMD solution by using the *rsync* tool.

Requirements for dual-feed handlers

The main goal of dual feed handlers is receiving data feeds from the source, and seamlessly retransmit the datagrams to the controller for error check. We think about two different approaches:

- Collecting all the datagrams before sending to the controller
- Retransmit each datagram to the controller instantly

We made experiments over the two solutions, and found that the second one affected the overall efficiency of the parallel system, as the instant retransmission will be trigger in the same sequence of the sender, and according to the fat-tree design of the system, the bandwidth of output from the controller is the aggregation of bandwidth over data handlers, therefore the maximum data frequency between the handler and the controller can only be a half of the incoming data.

Another analyse was performed by monitoring the verb queues in the MPI-CH architecture, and we found that the queuing buffer requirement for the second solution shows the similar result: the incoming data of frequency f , and the average size of the datagram is s , the size of verb queue buffer in the controller is approximately $2 \times \frac{s}{f}$ with no data loss, because the architecture of the controller is the same as the handlers, therefore the maximum frequency in verb queue consumption equals to that of handers, so the buffer size should be doubled. Along with the distribution strategy of filtering and error check discussed in earlier section, and our choice in the retransmit strategy, we can have our blue print for the dual-feed handlers

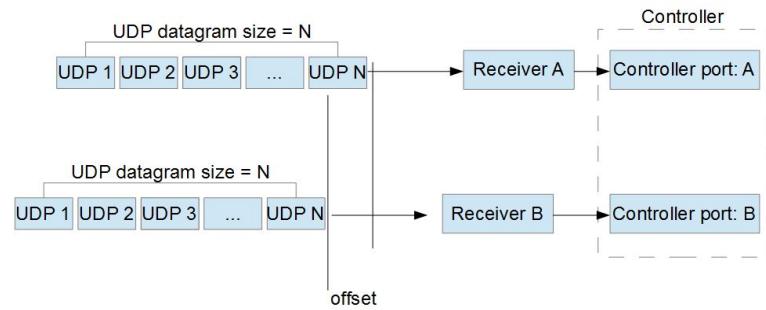


Figure 3.13: A-B data feed handler blueprint

Dual-feed handler implementation

The implementation of dual-feed handler is simple, and

3.6 FPGA Algorithm design

... design of the thread dispatching, event sensor in the master board

Chapter 4

Experiment Design, Requirements and Expectations

This chapter shows the design of experiment, requirements and expectations the project wants to achieve.

The main goal is to implement a data processor for the dual feed data sources using open-source hardware, and implement UDP package processing application on the FPGA for acceleration.

In order to achieve the goal of receiving dual feeds, which will come from physically two Ethernet cables, the system needs at least two Ethernet ports, and we introduces MPICH over multiple independent hardware which has one Ethernet port each, and congregate them into a cluster in order to meet the requirements.

4.1 Experiment Design

This section introduces the design of five experiments and their architectures:

Table 4.1: Experiment design table

Number	Name	Introduction
1	Single data feedwith single receiver	Normal UDP send/receive pair
2	Dual data feeds over the cluster	Using dual data feeds of normal UDP send/receiver pairs
3	Dual feeds processed with MPI processors without RDMA	Using dual data feeds sending data to MPI structure receivers without RDMA
4	Dual feeds processed with MPI processors with RDMA	Using dual data feeds sending data to MPI structure receivers with RDMA
5	Dual feeds processed with MPI with unified QoS check	Ignoring the QoS check of sockets when each node receives data, and performs an unified checksum after merging the feeds
6	Dual feeds processed with MPI with unified QoS check on FPGA	Ignoring the QoS check of sockets when each node receives data and performs an unified checksum after merging the feeds by FPGA

4.1.1 Experiment 1: Single data feed with single receiver

This experiment shows the behaviour of a single UDP sender(data source) and a single UDP receiver(client), and tracks the output of receiver in the following environment:

- No data lost in source and transmission
- Data lost in source
- Significant latency in data transmission
- Data source goes down

This experiment requires an computer simulating the data source, and a raspberry pi hosted receiver as the client. Quality of service strategy of data transmission is relied on the default settings of the client operating system.

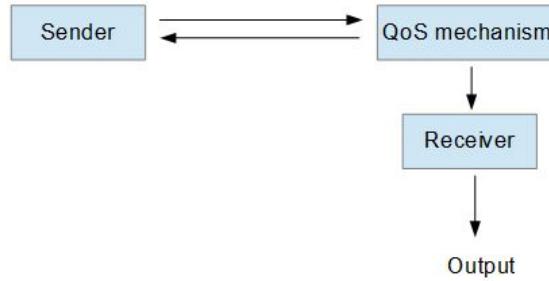


Figure 4.1: Design of experiment 1

4.1.2 Experiment 2: Dual data feeds over the cluster

This experiment shows the advantage of using dual feeds. Data source needs to simulate two data sources with the same content in package, and has an manual offset of one millisecond. The experiment needs to track the output from receivers under the following scenarios:

- No data lost in either sources
- One of the source simulates data lost, while the other does not
- Both sources have data lost
- One of the sources goes off-line, while the other one does not
- Both sources go off-line

The design is shown in picture 4.2.

Receiver 1 and 2 can communicate with each other using network protocols and contact $output_1$ and $output_2$ together to have an unified output, and shows the differences in data lost conditions.

The QoS mechanism is also relied on the configurations in operating system of the receiver, and the merging process will happen in the simulated master node.

4.1.3 Experiment 3: Dual feeds processed with MPI processors without RDMA

This experiment implements MPICH over the Raspberry Pi2 clusters, and the experiment shows output on the master node with dual data feed sources under the same scenarios as experiment2. The receiver programme communicate with each other using the message passing interfaces provided by MPICH, and shows the details of latency handling.

The MPICH program uses MPI_recv and MPI_send pair to communicate under the MPI framework, and the way it runs is using SSH client in the master to perform *mpiexec* shell

command, with the arguments of a file containing lists of servers, and configurations of pointing at the destination of each MPI programme.

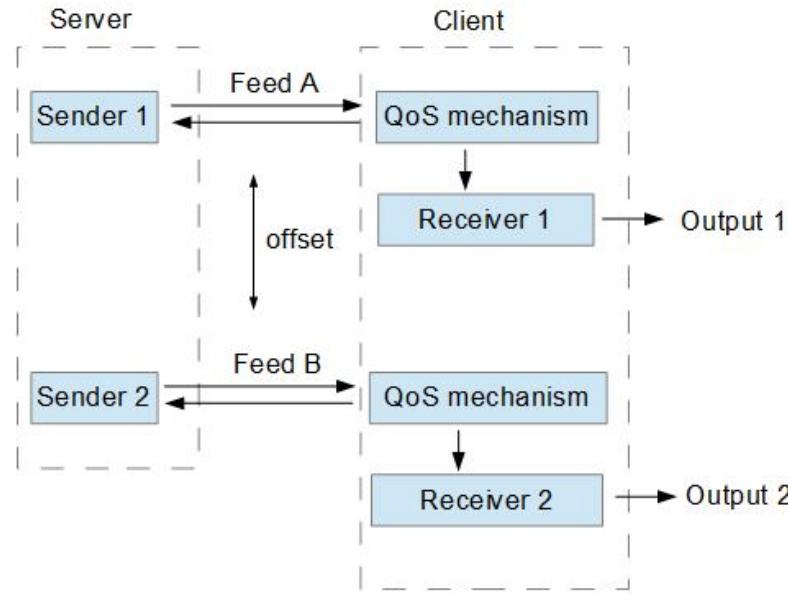


Figure 4.2: Design of experiment 2

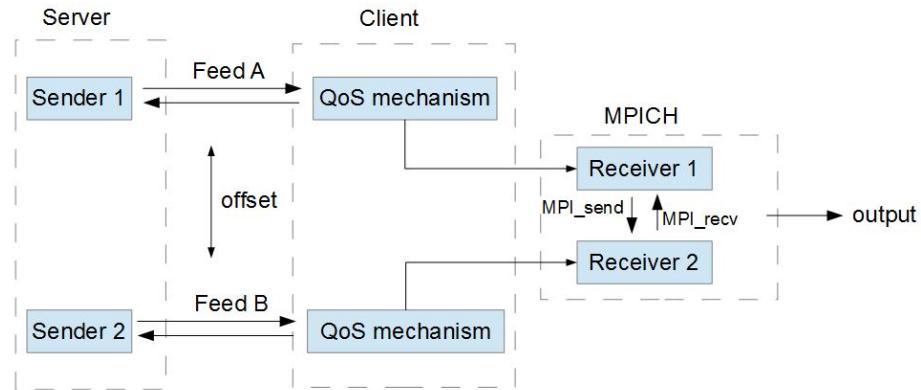


Figure 4.3: Design of experiment 3

Description

This experiment shows the

- UDP clients runs independently on two nodes, and passing the result to a master

node

- UDP clients

4.2 Experiment Requirements and Expectations

4.2.1 Simulation of the data source

An server needs to run on an separate computer to simulate two high-frequency data sources, and push the data via different programming interfaces to two destinations.

Chapter 5

Implementation and Testing

This is the chapter in which you review the implementation and testing decisions and issues, and critique these processes.

Code can be output inline using `\lstinline|some code|`. For example, this code is inline: `public static int example = 0;` (I have used the character `|` as a delimiter, but any non-reserved character not in the code text can be used.)

Code snippets can be output using the `\begin{lstlisting} ... \end{lstlisting}` environment with the code given in the environment. For example, consider listing 5.1, below.

Listing 5.1: Example code

```
public static void main() {  
    System.out.println("Hello -World");  
}
```

Code listings are produced using the package “Listings”. This has many useful options, so have a look at the package documentation for further ideas.

Chapter 6

Results

This is the chapter in which you review the outcomes, and critique the outcomes process. You may include user evaluation here too.

Chapter 7

Conclusions

This is the chapter in which you review the major achievements in the light of your original objectives, critique the process, critique your own learning and identify possible future work.

Bibliography

- Aho, A. V. and Hopcroft, J. E. (1974), *The design and analysis of computer algorithms*, Pearson Education India.
- Alexander, C. (2001), *Market models: A guide to financial data analysis*, John Wiley & Sons.
- Anderberg, M. R. (2014), *Cluster analysis for applications: probability and mathematical statistics: a series of monographs and textbooks*, Vol. 19, Academic press.
- Archer, C. J. and Blocksome, M. A. (2012), ‘Remote direct memory access’. US Patent 8,325,633.
- Arndt, R., Beukema, B., Craddock, D., Gregg, T., Schmidt, D. and Walk, B. (2003), ‘Infiniband subnet management queue pair emulation for multiple logical ports on a single physical port’. US Patent App. 10/626,988.
- Bedeir, T. (2010), Building an rdma-capable application with ib verbs, Technical report, Technical report, <http://www.hpcadvisorycouncil.com/>, August 2010. <http://www.hpcadvisorycouncil.com/pdf/building-an-rdma-capableapplication-with-ib-verbs.pdf>.
- Behrooz, P. (2005), ‘Computer architecture: From microprocessors to supercomputers’.
- Boyd, W. T., Joseph, D. J., Ko, M. A. and Recio, R. J. (2007), ‘Memory management offload for rdma enabled network adapters’. US Patent 7,299,266.
- Bradbury, A. (n.d.), ‘Raspberry pi spi’, <https://www.raspberrypi.org/documentation/configuration/raspi-config.md>. Accessed July 2, 2016.
- Brown, R. G. (2004), ‘Engineering a beowulf-style compute cluster’, *Duke University Physics Department*.
- Cáceres, R., Duffield, N. G., Horowitz, J., Towlsey, D. and Bu, T. (1999), Multicast-based inference of network-internal characteristics: Accuracy of packet loss estimation, in ‘INFOCOM’99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE’, Vol. 1, IEEE, pp. 371–379.

- Cisco (n.d.), ‘The next generation trading infrastructure’, http://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-3000-series-switches/white-paper_c11-720080.pdf. Accessed July 2, 2016.
- Clos, C. (1953), ‘A study of non-blocking switching networks’, *Bell System Technical Journal* **32**(2), 406–424.
- corp., R. (n.d.), ‘Realvnc on raspberry pi’, <https://www.realvnc.com/docs/raspberry-pi.html>. Accessed July 2, 2016.
- Cox, S. J., Cox, J. T., Boardman, R. P., Johnston, S. J., Scott, M. and Obrien, N. S. (2014), ‘Iridis-pi: a low-cost, compact demonstration cluster’, *Cluster Computing* **17**(2), 349–358.
- Darema, F. (2001), The spmd model: Past, present and future, in ‘European Parallel Virtual Machine/Message Passing Interface Users Group Meeting’, Springer, pp. 1–1.
- Dijkman, R., Hofstetter, J. and Koehler, J. (2011), *Business Process Model and Notation*, Springer.
- Dimitrov, R. and Skjellum, A. (1999), An efficient mpi implementation for virtual interface (vi) architecture-enabled cluster computing, in ‘Proceedings of the Third MPI Developers Conference’.
- Dongarra, J. J., Otto, S. W., Snir, M. and Walker, D. (1995), ‘An introduction to the mpi standard’, *Communications of the ACM* p. 18.
- Eggert, L. and Gont, F. (2009), ‘Tcp user timeout option’.
- Ellis, C. S. (1980), ‘Concurrent search and insertion in avl trees’, *IEEE Transactions on Computers* **100**(9), 811–817.
- FX, V. (n.d.), ‘Logi pi fpga user manual’, http://valentfx.com/wiki/index.php?title=Logi-Pi_Quick_Start_Guide. Accessed July 2, 2016.
- Garcia, D. J., Hilland, J. R., Culley, P. R., Barron, D. L. and Krause, M. R. (2006), ‘Binding a memory window to a queue pair’. US Patent 7,103,744.
- Gropp, W. and Lusk (1996), ‘A high-performance, portable implementation of the mpi message passing interface standard’, *Parallel computing* **22**(6), 789–828.
- Gropp, W. and Lusk, E. (1994), ‘An abstract device definition to support the implementation of a high-level point-to-point message-passing interface’, *Preprint MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill.*
- Harrington, W. (2015), *Learning Raspbian*, Packt Publishing Ltd.
- Hauck, S. and DeHon, A. (2010), *Reconfigurable computing: the theory and practice of FPGA-based computation*, Vol. 1, Morgan Kaufmann.

- Hilland, J., Culley, P., Pinkerton, J. and Recio, R. (2003), ‘Rdma protocol verbs specification’, *RDMAC Consortium Draft Specification draft-hilland-iwarp-verbsv1. 0-RDMAC*.
- Jain, V. (n.d.), ‘A low-latency solution for high frequency trading from ibm and mellanox’, <http://www.mellanox.com/pdf/whitepapers/Low-Latency-Solution-for-High-Frequency-Trading-from-IBM-and-Mellanox.pdf>. Accessed July 2, 2016.
- Junger, A., Moritz, R., Lueders, U., Thuss, W., Elbracht, B., Haensel, J. and Kostorz, W. (2007), ‘Method of data transmission and a transmission and reception device therefor’. US Patent 7,203,776.
- Kendall, W. (n.d.), ‘Mpif tutorial introduction mpi tutorial’, <http://mpitutorial.com/tutorials/mpi-introduction/>. Accessed July 2, 2016.
- Klaas, J. (2004), ‘System-on-a-chip’. US Patent 6,816,750.
- Kunz, W. E. and McCamey, A. (1999), ‘Rj-45 modular connector with microwave-transmission-line integrated signal conditioning for high speed networks’. US Patent 5,971,813.
- Laumonier, A. (n.d.), ‘Hft in my backyard’, <https://sniperinmahwah.wordpress.com/2014/09/22/hft-in-my-backyard-part-i/>. Accessed July 2, 2016.
- Le, D. (2009), ‘Automated trading system’.
- Leiserson, C. E. (1985), ‘Fat-trees: universal networks for hardware-efficient supercomputing’, *IEEE transactions on Computers* **100**(10), 892–901.
- Lin, H.-M., Huang, H.-J., Yuan, J.-m. and Bai, M.-C. (2008), ‘Transmission device and data extended transmission method’. US Patent App. 12/203,007.
- Liu, J., Wu, J. and Panda, D. K. (2004), ‘High performance rdma-based mpi implementation over infiniband’, *International Journal of Parallel Programming* **32**(3), 167–198.
- Michael, M. S. (1992), ‘Universal asynchronous receiver/transmitter’. US Patent 5,140,679.
- Moon, S. B., Kurose, J., Skelly, P. and Towsley, D. (1998), Correlation of packet delay and loss in the internet, Technical report, Technical report, UMass CMPSCI Technical Report 98-11.
- Morris, G. W., Thomas, D. B. and Luk, W. (2009), Fpga accelerated low-latency market data feed processing, in ‘2009 17th IEEE Symposium on High Performance Interconnects’, IEEE, pp. 83–89.
- MPI-CH (n.d.), ‘Mpich documentation’, https://wiki.mpich.org/mpich/index.php/CH3_And_Channels. Accessed July 2, 2016.

- Org, M.-C. (n.d.a), ‘Mpich documentation’, http://www.mpich.org/static/docs/v3.2/www3/MPI_Init.html. Accessed July 2, 2016.
- Org., R. (n.d.b), ‘Raspberry pi spi’, <https://www.raspberrypi.org/documentation/hardware/raspberrypi/spi/README.md>. Accessed July 2, 2016.
- org., R. P. (n.d.c), ‘Vnc on raspberry pi’, <https://www.raspberrypi.org/documentation/remote-access/vnc/>. Accessed July 2, 2016.
- org, T. M. (n.d.d), ‘Mpi4py documentation’, <https://pythonhosted.org/mpi4py/usrman/index.html>. Accessed July 2, 2016.
- Pang, Z., Xie, M., Zhang, J., Zheng, Y., Wang, G., Dong, D. and Suo, G. (2014), ‘The express high performance interconnect networks’, *Frontiers of Computer Science* **8**(3), 357–366.
- Postel, J. (n.d.), ‘Rfc 768 user datagram protocol’, <https://tools.ietf.org/html/rfc768>. Accessed July 2, 2016.
- Protopopov, B. V. and Skjellum, A. (2001), ‘A multithreaded message passing interface (mpi) architecture: Performance and program issues’, *Journal of Parallel and Distributed Computing* **61**(4), 449–466.
- register .Inc, T. (n.d.), ‘Review: Beagleboard beaglebone black’, http://www.theregister.co.uk/2013/06/11/review_beagleboard_beaglebone_black/. Accessed July 2, 2016.
- Russell, R. D. (2012), ‘Introduction to rdma programming’. US Patent App. 10/929,943.
- Schwartz, J. T. (1980), ‘Ultracomputers’, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **2**(4), 484–521.
- Stone, H. S. (1971), ‘Parallel processing with the perfect shuffle’, *IEEE transactions on computers* **20**(2), 153–161.
- Stout, Q. F. and Warren, B. L. (1986), ‘Tree rebalancing in optimal time and space’, *Communications of the ACM* **29**(9), 902–908.
- ValentFx (n.d.a), ‘Logi pi wishbone document’, http://valentfx.com/wiki/index.php?title=LOGI_-_Wishbone_-_HW. Accessed July 2, 2016.
- ValentFx, T. (n.d.b), ‘Logi pi user manual’, http://valentfx.com/wiki/index.php?title=LOGI_Pi_User_Manual. Accessed July 2, 2016.
- ValentFx, T. (n.d.c), ‘Logi pi wishbone hardware documentation’, http://valentfx.com/wiki/index.php?title=LOGI_-_Wishbone_-_HW. Accessed July 2, 2016.
- Yahoo! (n.d.), ‘Yahoo! finance api’, <http://finance.yahoo.com/quote/API?ltr=1>. Accessed July 2, 2016.

- Yang, X.-J., Liao, X.-K., Lu, K., Hu, Q.-F., Song, J.-Q. and Su, J.-S. (2011), 'The tianhe-1a supercomputer: its hardware and software', *Journal of Computer Science and Technology* **26**(3), 344–351.
- Zusman, J., Tang, J. L., Nakelsky, R. S., Verbeck, S. L. and Azizian, D. (1999), 'Fault-tolerant central ticker plant system for distributing financial market data'. US Patent 5,987,432.

Appendix A

Design Diagrams

Appendix B

User Documentation

Appendix C

Raw results output

Appendix D

Code

D.1 File: yourCodeFile.java

```
// This is an example java code file , just for  
illustration purposes
```

```
public static void main() {  
    System.out.print ("Hello_World");  
}
```